RogueShooter – All Scripts

Generated: 2025-11-11 08.58 UTC
Files: 144
Scanned: Assets/scripts

Assets/scripts/Boot/OfflineSceneBoot.cs

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

public static class OfflineSceneBoot
{
    // Ajetaan aina, heti kun ensimmäinen scene on ladattu (buildissä Core)
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterSceneLoad)]
    static void EnsureOfflineLevel()
    {
        if (NetMode.IsOnline) return;

        // Jos jokin muu kuin Core on jo auki (esim. online host vaihtoi scenen), älä tee mitään
        for (int i = 0; i < SceneManager.sceneCount; i++)
        {
            var s = SceneManager.GetSceneAt(i);
            if (s.isLoaded && s.name != (LevelLoader.Instance?.CoreSceneName ?? "Core"))
                return;
        }

        // Offline-käynnistys: ei tukeuduta Mirrorin tilaan ollenkaan
        var _ = CoroutineRunner.Run(Co_Boot());
    }

    static IEnumerator Co_Boot()
    {
        // 1) Core ladattuna ja aktiiviseksi
        string coreName = LevelLoader.Instance?.CoreSceneName ?? "Core";

        var core = SceneManager.GetSceneByName(coreName);
        if (!core.IsValid() || !core.isLoaded)
        {
            var loadCore = SceneManager.LoadSceneAsync(coreName, LoadSceneMode.Additive);
            while (loadCore != null && !loadCore.isDone) yield return null;
            core = SceneManager.GetSceneByName(coreName);
        }
        if (core.IsValid()) SceneManager.SetActiveScene(core);

        // 2) Päätä ladattava kenttä LevelLoaderista
        string requested =
            LevelLoader.Instance?.CurrentLevel ??
            LevelLoader.Instance?.DefaultLevel;

        // Fallback: vain jos “Level 0” on oikeasti Build Settingsissä
        if (string.IsNullOrEmpty(requested) || !Application.CanStreamedLevelBeLoaded(requested))
        {
            if (!string.IsNullOrEmpty(requested))
                Debug.LogWarning($"[OfflineBoot] '{requested}' ei ole Build Settingsissä. Yritetään fallbackia.");

            if (Application.CanStreamedLevelBeLoaded("Level 0"))
```

```
            requested = "Level 0";
        else
        {
            Debug.LogError("[OfflineBoot] Ei löydy ladattavaa kenttää: Current/Default puuttuu ja 'Level 0' ei ole Build Settingsissä.");
            yield break;
        }
    }

    var op = SceneManager.LoadSceneAsync(requested, LoadSceneMode.Additive);
    while (op != null && !op.isDone) yield return null;

    var map = SceneManager.GetSceneByName(requested);
    if (!map.IsValid() || !map.isLoaded)
    {
        Debug.LogError($"[OfflineBoot] Scene '{requested}' ei latautunut.");
        yield break;
    }

    // 4) Aktivoi map yhdeksi frameksi, jotta sen Start/OnEnable ehtivät
    SceneManager.SetActiveScene(map);
    yield return null;                  // 1 frame
    yield return new WaitForEndOfFrame();

    // 5) Kenttäkohtaiset hookit (esim. edge bake, miehitys)
    var edgeBaker = Object.FindFirstObjectByType<EdgeBaker>();
    edgeBaker?.BakeAllEdges();

    if (LevelGrid.Instance != null)
        LevelGrid.Instance.RebuildOccupancyFromScene();

    MousePlaneMap.Instance.Rebuild();
    // 6) Palauta Core aktiiviseksi (UI yms.) ja ilmoita, että level on valmis
    if (core.IsValid()) SceneManager.SetActiveScene(core);

    try { LevelLoader.RaiseLevelReady(map); } catch { /* ei kriittinen */ }

}

// Minimaalinen “global coroutine host” ilman mitään GameObjectia
private sealed class CoroutineRunner : MonoBehaviour
{
    static CoroutineRunner _inst;
    public static Coroutine Run(IEnumerator e)
    {
        if (_inst == null)
        {
            var go = new GameObject("~OfflineBoot");
            Object.DontDestroyOnLoad(go);
            _inst = go.AddComponent<CoroutineRunner>();
        }
        return _inst.StartCoroutine(e);
    }
```

```
    }
}
```

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
{
    public static CameraManager Instance { get; private set; }
    [SerializeField] private GameObject actionCameraGameObject;

    [SerializeField] private float actionCameraVerticalPosition = 2.5f;

    [SerializeField] private bool actionCameraOn = true;

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

        Instance = this;
    }

    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);
    }

    public 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();

                if (shooterUnit == null || targetUnit == null)
                {
                    Debug.LogWarning("[CameraManager] Cannot setup action camera: shooter or target is null");
                    return;
                }

                Vector3 cameraCharacterHeight = Vector3.up * actionCameraVerticalPosition;
                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);

                if (actionCameraOn)
                {
                    ShowActionCamera();
                }

                break;
        }
    }

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

}
```

Assets/scripts/Camera/CameraThaw.cs

```
// Assets/Scripts/GameLogic/Camera/CameraThaw.cs
using UnityEngine;

public static class CameraThaw
{
    public static void Thaw(string reason = "")
    {
        Debug.Log($"[CameraThaw] Thaw camera. Reason={reason}");

        // 1) Poista mahdollinen action-kamera pelistä
        var cm = CameraManager.Instance;    // teillä jo oleva manageri, joka togglaa action-kameran
        if (cm != null) cm.HideActionCamera(); // no-op jos ei ollut päällä

        // 2) Varmista vapaa kamera päällä
        var cc = CameraController.Instance;
        if (cc != null) cc.enabled = true;

        // 3) Pakota input-map takaisin päälle
        var im = InputManager.Instance;
        if (im != null) im.ForceEnablePlayerMap();
    }
}
```

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);
    }

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

    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; }

    [SerializeField]
    private CinemachineImpulseSource cinemachineRecoilImpulseSource;

    [SerializeField]
    private CinemachineImpulseSource cinemachineExplosiveImpulseSource;

    [Header("Recoil Shake Settings")]
    [Tooltip("Minimum time between recoil shakes to prevent stacking")]
    [SerializeField]
    private float recoilCooldown = 0.1f;

    private float lastRecoilTime = -999f;

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

        Instance = this;
    }

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

    public void RecoilCameraShake(float ShakeStrength)
    {
        if (Time.time - lastRecoilTime < recoilCooldown)
        {
            return;
        }

        lastRecoilTime = Time.time;
        cinemachineRecoilImpulseSource.GenerateImpulse(ShakeStrength);
    }
}
```

Assets/scripts/Camera/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;
         MeleeAction.OnAnyMeleeActionHit -= MeleeAction_OnAnyMeleeActionHit;
    }

    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/Combat/BulletTargetCalculator.cs

```
using UnityEngine;

public static class BulletTargetCalculator
{
    [System.Serializable]
    public class HitZoneConfig
    {
        [Header("Critical Hit Zones (Head/Heart)")]
        public float critHeightMin = 2.2f;
        public float critHeightMax = 2.8f;
        public float critHorizontalSpread = 0.15f;

        [Header("Normal Hit Zone (Torso)")]
        public float hitHeightMin = 1.5f;
        public float hitHeightMax = 2.2f;
        public float hitHorizontalSpread = 0.25f;

        [Header("Graze Zones (Limbs)")]
        public float grazeHeightMin = 0.8f;
        public float grazeHeightMax = 2.5f;
        public float grazeHorizontalSpread = 0.4f;

        [Header("Close Call (Near Miss)")]
        public float closeHorizontalOffset = 0.5f;
        public float closeVerticalSpread = 0.3f;

        [Header("Critical Miss (Wild Shot)")]
        public float critMissHorizontalOffset = 1.5f;
        public float critMissVerticalSpread = 1.0f;
    }

    private static HitZoneConfig _config;
    public static HitZoneConfig Config
    {
        get
        {
            if (_config == null)
            {
                _config = new HitZoneConfig();
            }
            return _config;
        }
        set => _config = value;
    }

    public static Vector3 CalculateBulletTarget(Unit targetUnit, ShotTier tier, Unit shootingUnit = null)
    {
        if (targetUnit == null) return Vector3.zero;

        Vector3 basePosition = targetUnit.GetWorldPosition();
```

```
        var cfg = Config;

        switch (tier)
        {
            case ShotTier.Crit:
                return CalculateCriticalHit(basePosition, cfg);

            case ShotTier.Hit:
                return CalculateNormalHit(basePosition, cfg);

            case ShotTier.Graze:
                return CalculateGraze(basePosition, cfg);

            case ShotTier.Close:
                return CalculateCloseMiss(basePosition, shootingUnit, cfg);

            case ShotTier.CritMiss:
                return CalculateCriticalMiss(basePosition, shootingUnit, cfg);

            default:
                return basePosition + Vector3.up * 1.5f;
        }
    }

    private static Vector3 CalculateCriticalHit(Vector3 basePos, HitZoneConfig cfg)
    {
        float height = Random.Range(cfg.critHeightMin, cfg.critHeightMax);
        float xOffset = Random.Range(-cfg.critHorizontalSpread, cfg.critHorizontalSpread);
        float zOffset = Random.Range(-cfg.critHorizontalSpread, cfg.critHorizontalSpread);

        return basePos + new Vector3(xOffset, height, zOffset);
    }

    private static Vector3 CalculateNormalHit(Vector3 basePos, HitZoneConfig cfg)
    {
        float height = Random.Range(cfg.hitHeightMin, cfg.hitHeightMax);
        float xOffset = Random.Range(-cfg.hitHorizontalSpread, cfg.hitHorizontalSpread);
        float zOffset = Random.Range(-cfg.hitHorizontalSpread, cfg.hitHorizontalSpread);

        return basePos + new Vector3(xOffset, height, zOffset);
    }

    private static Vector3 CalculateGraze(Vector3 basePos, HitZoneConfig cfg)
    {
        bool hitLimb = Random.value > 0.5f;

        float height;
        if (hitLimb)
        {
            height = Random.value > 0.5f
                ? Random.Range(cfg.grazeHeightMin, cfg.hitHeightMin)
                : Random.Range(cfg.hitHeightMax, cfg.grazeHeightMax);
```

```
        }
        else
        {
            height = Random.Range(cfg.grazeHeightMin, cfg.grazeHeightMax);
        }

        float xOffset = Random.Range(-cfg.grazeHorizontalSpread, cfg.grazeHorizontalSpread);
        float zOffset = Random.Range(-cfg.grazeHorizontalSpread, cfg.grazeHorizontalSpread);

        return basePos + new Vector3(xOffset, height, zOffset);
    }

    private static Vector3 CalculateCloseMiss(Vector3 basePos, Unit shootingUnit, HitZoneConfig cfg)
    {
        Vector3 direction = Vector3.right;

        if (shootingUnit != null)
        {
            Vector3 toTarget = basePos - shootingUnit.GetWorldPosition();
            toTarget.y = 0;
            if (toTarget.sqrMagnitude > 0.001f)
            {
                Vector3 perpendicular = Vector3.Cross(toTarget.normalized, Vector3.up);
                direction = Random.value > 0.5f ? perpendicular : -perpendicular;
            }
        }
        else
        {
            direction = Random.insideUnitCircle.normalized;
            direction = new Vector3(direction.x, 0, direction.y);
        }

        float offset = Random.Range(cfg.closeHorizontalOffset * 0.7f, cfg.closeHorizontalOffset);
        float height = Random.Range(cfg.hitHeightMin, cfg.hitHeightMax) +
                       Random.Range(-cfg.closeVerticalSpread, cfg.closeVerticalSpread);

        return basePos + direction * offset + Vector3.up * height;
    }

    private static Vector3 CalculateCriticalMiss(Vector3 basePos, Unit shootingUnit, HitZoneConfig cfg)
    {
        Vector3 randomDir = Random.insideUnitCircle.normalized;
        Vector3 direction = new Vector3(randomDir.x, 0, randomDir.y);

        float offset = Random.Range(cfg.critMissHorizontalOffset * 0.8f, cfg.critMissHorizontalOffset * 1.2f);
        float height = Random.Range(0.5f, 3.0f) + Random.Range(-cfg.critMissVerticalSpread, cfg.critMissVerticalSpread);

        return basePos + direction * offset + Vector3.up * height;
    }
}
```

Assets/scripts/Combat/CombatRanges.cs

```
using UnityEngine;

[CreateAssetMenu(menuName="RogueShooter/Combat Ranges")]
public class CombatRanges : ScriptableObject
{

    [Header("Use tiles instead of world units")]
    public bool useTiles = true;

    [Header("Max distance per band (in tiles)")]
    public int meleeMaxTiles  = 1;  // "vieressä"
    public int closeMaxTiles  = 5;
    public int mediumMaxTiles = 15;
    public int longMaxTiles = 20;

    [Header("Legacy world units (fallback if useTiles==false)")]
    public float meleeMaxWU  = 1.2f;
    public float closeMaxWU  = 4f;
    public float mediumMaxWU = 9f;
    public float longMaxWU   = 15f;
}
```

Assets/scripts/Combat/ShootingResolver.cs

```
using UnityEngine;

public struct ShotResult {
    public ShotTier tier;
    public int damage;
    public bool bypassCover;
    public bool coverOnly;
}

public static class ShootingResolver
{
    private static CombatRanges _cachedRanges;
    private static CombatRanges Ranges
    {
        get
        {
            if (!_cachedRanges)
                _cachedRanges = Resources.Load<CombatRanges>("CombatRanges");
            return _cachedRanges;
        }
    }

    private static float TileDistance(GridPosition a, GridPosition b)
    {
        int dx = Mathf.Abs(a.x - b.x);
        int dz = Mathf.Abs(a.z - b.z);
        int cost = SircleCalculator.Sircle(dx, dz);
        return cost / 10f;
    }

    public static RangeBand GetBandTiles(Unit attacker, Unit target, WeaponDefinition w)
    {
        var gpA = attacker.GetGridPosition();
        var gpT = target.GetGridPosition();

        if (Ranges && Ranges.useTiles)
        {
            float tiles = TileDistance(gpA, gpT);
            if (tiles <= Ranges.meleeMaxTiles)  return RangeBand.Melee;
            if (tiles <= Ranges.closeMaxTiles)  return RangeBand.Close;
            if (tiles <= Ranges.mediumMaxTiles) return RangeBand.Medium;
            if (tiles <= Ranges.longMaxTiles)   return RangeBand.Long;
            return RangeBand.Extreme;
        }

        Vector3 aw = attacker.GetWorldPosition();
        Vector3 tw = target.GetWorldPosition();
        float distWU = Vector3.Distance(aw, tw);
        if (Ranges)
        {
```

```
            if (distWU <= Ranges.meleeMaxWU)  return RangeBand.Melee;
            if (distWU <= Ranges.closeMaxWU)  return RangeBand.Close;
            if (distWU <= Ranges.mediumMaxWU) return RangeBand.Medium;
            if (distWU <= Ranges.longMaxWU)   return RangeBand.Long;
            return RangeBand.Extreme;
        }

        if (distWU <= 1.2f)          return RangeBand.Melee;
        if (distWU <= w.closeMax)    return RangeBand.Close;
        if (distWU <= w.mediumMax)   return RangeBand.Medium;
        if (distWU <= w.longMax)     return RangeBand.Long;
        return RangeBand.Extreme;
    }

    public static ShotResult Resolve(Unit attacker, Unit target, WeaponDefinition w, bool isOverwatchShot = false)
    {
        Vector3 a = attacker.GetWorldPosition();
        Vector3 t = target.GetWorldPosition();
        float dist = Vector3.Distance(a, t);
        var band = GetBandTiles(attacker, target, w);

        int baseHit = GetBaseHitChance(band, w);
        baseHit += GetSkillBonus(attacker);
        Debug.Log("BaseHit: " + baseHit);

        float hitChanceMultiplier = GetHitPenaltyMultiplier(attacker, target, w, isOverwatchShot);
        baseHit = Mathf.RoundToInt(baseHit * hitChanceMultiplier);

        Debug.Log("Basehit - Overall Penalty = " + baseHit);
        baseHit = Mathf.Clamp(baseHit, 0, 100);

        int roll1 = UnityEngine.Random.Range(1, 101);
        bool isHit = roll1 <= baseHit;

        ShotTier tier = isHit
            ? RollOnHit(band, w)
            : ShotTier.CritMiss;

        var res = new ShotResult { tier = tier };
        ApplyDamageModel(ref res, w);

       // DebugShot(attacker, target, w, band, baseHit, roll1, res);
        return res;
    }

    private static int GetBaseHitChance(RangeBand b, WeaponDefinition w)
    {
        if (w.useAdvancedAccuracy)
            return w.GetTuning(b).baseHitChance;

        switch (b)
        {
```

```
            case RangeBand.Melee:  return w.meleeAcc;
            case RangeBand.Close:  return w.closeAcc;
            case RangeBand.Medium: return w.mediumAcc;
            case RangeBand.Long:   return w.longAcc;
            default:               return w.extremeAcc;
        }
    }

    private static int GetSkillBonus(Unit attacker)
    {
        if (attacker != null && attacker.archetype != null)
            return attacker.archetype.shootingSkillLevel * attacker.archetype.accuracyBonusPerSkillLevel;
        return 0;
    }

    private static float GetHitPenaltyMultiplier(Unit attacker, Unit target, WeaponDefinition weapon, bool isOverwatchShot)
    {
        if (target == null || target.archetype == null)
            return 1f;

        float totalPenaltyPercent = 0f;
        var targetArch = target.archetype;

        var targetGridPosition = target.GetGridPosition();
        var node = PathFinding.Instance.GetNode(targetGridPosition.x, targetGridPosition.z, targetGridPosition.floor);
        var coverType = CoverService.EvaluateCoverHalfPlane(attacker.GetGridPosition(), target.GetGridPosition(), node);

        if (coverType == CoverService.CoverType.High)
            totalPenaltyPercent += targetArch.highCoverEnemyHitPenalty;
        else if (coverType == CoverService.CoverType.Low)
            totalPenaltyPercent += targetArch.LowCoverEnemyHitPenalty;

        Debug.Log("CoverPenalty: " + totalPenaltyPercent);

        var moveAction = target.GetAction<MoveAction>();
        if (moveAction != null && moveAction.IsActionActive())
        {
            totalPenaltyPercent += targetArch.moveEnemyHitPenalty;

            var movePenalty = targetArch.moveEnemyHitPenalty;
            Debug.Log("MovePenalty: " + movePenalty);
        }

        if (isOverwatchShot)
        {
            totalPenaltyPercent += weapon.overwatch.overwatchShootPenalty;
            var overwachShot = weapon.overwatch.overwatchShootPenalty;
            Debug.Log("OverwachShotPenalty : " + overwachShot);
        }

        return 1f - (totalPenaltyPercent / 100f);
    }
```

```
    private static ShotTier RollOnHit(RangeBand b, WeaponDefinition w)
    {
        var t = w.GetTuning(b);

        int close = Mathf.Max(0, t.onHit_Close);
        int graze = Mathf.Max(0, t.onHit_Graze);
        int hit   = Mathf.Max(0, t.onHit_Hit);
        int crit  = Mathf.Max(0, t.onHit_Crit);

        int sum = close + graze + hit + crit;
        if (sum <= 0) return ShotTier.Hit;

        int r = UnityEngine.Random.Range(1, sum + 1);
        if (r <= close) return ShotTier.Close;
        r -= close;
        if (r <= graze) return ShotTier.Graze;
        r -= graze;
        if (r <= hit) return ShotTier.Hit;
        return ShotTier.Crit;
    }

    private static void ApplyDamageModel(ref ShotResult res, WeaponDefinition w)
    {
        switch (res.tier)
        {
            case ShotTier.CritMiss:
                res.damage = 0;
                res.coverOnly = false;
                res.bypassCover = false;
                break;

            case ShotTier.Close:
                res.damage = Mathf.RoundToInt(w.baseDamage * w.missChipFactor);
                res.coverOnly = true;
                res.bypassCover = false;
                break;

            case ShotTier.Graze:
                res.damage = Mathf.RoundToInt(w.baseDamage * w.grazeFactor);
                res.coverOnly = false;
                res.bypassCover = false;
                break;

            case ShotTier.Hit:
                res.damage = w.baseDamage;
                res.coverOnly = false;
                res.bypassCover = false;
                break;

            case ShotTier.Crit:
                res.damage = w.baseDamage + w.critBonusDamage;
```

```
                res.coverOnly = false;
                res.bypassCover = true;
                break;
        }
    }

    private static void DebugShot(Unit attacker, Unit target, WeaponDefinition w, RangeBand band, int baseHit, int roll1, ShotResult result)
    {
        string tierColor =
            result.tier == ShotTier.Crit ? "Green" :
            result.tier == ShotTier.Hit ? "Blue" :
            result.tier == ShotTier.Graze ? "yellow" :
            result.tier == ShotTier.Close ? "orange" : "red";

        string txt =
            $"<b>{attacker.name}</b> → <b>{target.name}</b>\n" +
            $"Weapon: {w.name}\n" +
            $"Range: {band} | Roll1: {roll1} vs Hit%:{baseHit}\n" +
            $"Result: <color={tierColor}>{result.tier}</color> | Dmg:{result.damage} | " +
            $"{(result.bypassCover ? "Bypass Cover" : result.coverOnly ? "Cover Only" : "Normal")}";

        Debug.Log(txt);
    }
}
```

Assets/scripts/Combat/Weapons/Grenade/ArcApexSolver.cs

```
using UnityEngine;

public static class ArcApexSolver
{
    public static float ComputeCeilingClampedApex(
        Vector3 start, Vector3 end,
        ThrowArcConfig cfg,
        LayerMask ceilingMask,
        float ceilingClearance = 0.08f,
        int samples = 16,
        float? farWUOverride = null,
        AnimationCurve fallback = null,
        float targetAboveClearance = 0.05f )
    {
        float dWU = Vector2.Distance(new Vector2(start.x, start.z), new Vector2(end.x, end.z));
        float farWU = farWUOverride ?? (cfg != null ? cfg.farRangeWU : dWU);

        float apexNominal = (cfg != null)
        ? cfg.EvaluateApex(dWU, farWU)
        : Mathf.Lerp(7f, 1.2f, Mathf.Clamp01(dWU / 12f));

        if (apexNominal <= 0f) return 0f;

        var curve = (cfg != null && cfg.arcYCurve != null) ? cfg.arcYCurve : (fallback ?? AnimationCurve.Linear(0,0,1,0));

        float maxHeadroom = Mathf.Max(start.y, end.y) + apexNominal;

        float apexCap = float.PositiveInfinity;

        for (int i = 1; i <= samples; i++)
        {
            float t = i / (samples + 1f);
            Vector3 baseP = Vector3.Lerp(start, end, t);
            float baselineY = Mathf.Lerp(start.y, end.y, t);

            // Rajoita raycastin pituus: etsi katto vain relevantin alueen sisältä
            float maxRayDistance = Mathf.Max(5f, maxHeadroom - baseP.y + 1f);

            if (Physics.Raycast(baseP + Vector3.up * 0.01f, Vector3.up, out var hit, maxRayDistance, ceilingMask, QueryTriggerInteraction.Collide))
            {
                // UUSI EHTO: Hyväksy katto vain jos se on reitin "yläpuolella"
                // eli korkeammalla kuin sekä lähtö- että maalipiste
                float minRelevantHeight = Mathf.Max(start.y, end.y);
                if (hit.point.y <= minRelevantHeight + targetAboveClearance)
                    continue;

                float allowedArc = (hit.point.y - ceilingClearance) - baselineY;

                // Jos katto on liian matalalla (negatiivinen allowedArc), ohita
                if (allowedArc <= 0f)
```

```
                    continue;

                float c = Mathf.Max(1e-4f, curve.Evaluate(t));
                float localCap = allowedArc / c;
                if (localCap < apexCap) apexCap = localCap;
            }
        }

        if (float.IsPositiveInfinity(apexCap))
            return apexNominal;

        return Mathf.Max(0f, Mathf.Min(apexNominal, apexCap));
    }
}
```

Assets/scripts/Combat/Weapons/Grenade/ArcMath.cs

```
using UnityEngine;

public static class ArcMath
{
    public static float ComputeDescentAngleDeg(
        Vector3 start, Vector3 end,
        ThrowArcConfig cfg,
        float lift = 0f,
        AnimationCurve fallback = null)
    {
        // Sama apex ja käyrä kuin projektiilissa/previewssa
        float dWU = Vector2.Distance(new Vector2(start.x, start.z), new Vector2(end.x, end.z));
        float apexWU = (cfg != null) ? cfg.EvaluateApex(dWU, cfg.farRangeWU) : Mathf.Lerp(7f, 1.2f, Mathf.Clamp01(dWU / 12f));
        var curve = (cfg != null && cfg.arcYCurve != null) ? cfg.arcYCurve : (fallback ?? AnimationCurve.Linear(0,0,1,0));

        float tA = cfg ? Mathf.Clamp01(cfg.angleSampleT1) : 0.92f;
        float tB = cfg ? Mathf.Clamp01(cfg.angleSampleT2) : 0.98f;

        Vector3 pA = GetArcPoint(start, end, curve, apexWU, tA, lift);
        Vector3 pB = GetArcPoint(start, end, curve, apexWU, tB, lift);

        float dy = pA.y - pB.y; // positiivinen = laskeutuu alaspäin
        float dPlanar = Vector2.Distance(new Vector2(pA.x, pA.z), new Vector2(pB.x, pB.z));
        if (dPlanar < 1e-4f) return 90f; // käytännössä pystysuora

        return Mathf.Atan2(Mathf.Abs(dy), dPlanar) * Mathf.Rad2Deg;
    }

    private static Vector3 GetArcPoint(Vector3 start, Vector3 end, AnimationCurve curve, float apexWU, float t, float lift)
    {
        Vector3 p = Vector3.Lerp(start, end, t);
        float baselineY = Mathf.Lerp(start.y, end.y, t);
        p.y = baselineY + curve.Evaluate(t) * apexWU + lift;
        return p;
    }
}
```

Assets/scripts/Combat/Weapons/Grenade/ArcVisibility.cs

```
using UnityEngine;

public static class ArcVisibility
{
    /// True jos kaari mahtuu start→end osumatta “riittävän korkeisiin” esteisiin.
    public static bool IsArcClear(
        Vector3 start,
        Vector3 end,
        ThrowArcConfig cfg,
        int segments,
        LayerMask mask,
        Transform ignoreRoot = null,
        float lift = 0.2f,              // pieni nosto irti lattiasta
        float capsuleRadius = 12f,    // “paksuus” (kranaatin säde)
        float heightClearance = 0.05f,  // toleranssi
        float tStart = 0.02f,           // leikkaa alku- ja loppukärjet pois
        float tEnd = 0.98f,
        AnimationCurve fallbackCurve = null,
        float cellSizeWU = 2f,
        float fullTilePerc = 0.8f,
        float tallWallY = 0.6f,
        float? apexOverrideWU = null        // ← UUSI
        )

    {

        if (segments < 2) segments = 2;

    // Apex: käytä overridea, muutoin nominaali
        float dWU = Vector2.Distance(new Vector2(start.x, start.z), new Vector2(end.x, end.z));
        float apexWU = apexOverrideWU.HasValue
            ? apexOverrideWU.Value
            : (cfg != null ? cfg.EvaluateApex(dWU, cfg.farRangeWU)
                        : Mathf.Lerp(7f, 1.2f, Mathf.Clamp01(dWU / 12f)));

        var curve = (cfg != null && cfg.arcYCurve != null) ? cfg.arcYCurve : (fallbackCurve ?? AnimationCurve.Linear(0,0,1,0));

        Transform root = ignoreRoot ? ignoreRoot.root : null;

        for (int i = 0; i < segments; i++)
        {
            float t0 = Mathf.Lerp(tStart, tEnd,  i      / (float)segments);
            float t1 = Mathf.Lerp(tStart, tEnd, (i + 1) / (float)segments);

            Vector3 p0 = GetArcPoint(start, end, curve, apexWU, t0, lift);
            Vector3 p1 = GetArcPoint(start, end, curve, apexWU, t1, lift);

            // Keskiarvokorkeus päätökseen “ylittääkö”
            float segMidY = 0.5f * (p0.y + p1.y);
```

```
            // Hae kaikki collidereiden osumat kapselilla (havaitsee myös “start inside”)
            var hits = Physics.OverlapCapsule(p0, p1, capsuleRadius, mask, QueryTriggerInteraction.Collide);
            if (hits == null || hits.Length == 0) continue;

            foreach (var col in hits)
            {
                if (!col) continue;
                if (root && col.transform.root == root) continue; // ohita oma hahmo

                // Jos esteen yläreuna on segmentin korkeuden tasalla tai yli → blokkaa
                float topY = col.bounds.max.y; // AABB, mutta riittää konservatiiviseksi päätökseksi
                if (topY >= segMidY - heightClearance)
                    return false;
            }
        }

        return true;
    }

    private static Vector3 GetArcPoint(Vector3 start, Vector3 end, AnimationCurve curve, float apexWU, float t, float lift)
    {
        Vector3 p = Vector3.Lerp(start, end, t);
        float baselineY = Mathf.Lerp(start.y, end.y, t);
        p.y = baselineY + curve.Evaluate(t) * apexWU + lift;
        return p;
    }
}
```

Assets/scripts/Combat/Weapons/Grenade/GranadeProjectile.cs

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

public class GrenadeProjectile : NetworkBehaviour
{
   // ====================== DETERMINISTIC DEFLECTION (OFFLINE) ======================
    [Header("Deflection: Toggle & Mask")]
    [Tooltip("Kytkee deterministisen kimpoilun päälle vain offline-tilassa.")]
    [SerializeField] private bool useDeterministicDeflectionOffline = false;

    [Tooltip("Esineet ja asiat, joista kimpoillaan (seinät/ovet).")]
    [SerializeField] private LayerMask deflectionMask;

    [Header("Floor bounce randomization (OFFLINE visuals only)")]
    [Tooltip("Sallitun lattia-pompun pituuden haarukka WU:na (per throw).")]
    [SerializeField] private Vector2 floorBounceDistanceRangeWU = new Vector2(0.18f, 0.35f);

    [Tooltip("Minimipituuden haarukka WU:na, ettei pomppu typähdä nollaan (per throw).")]
    [SerializeField] private Vector2 minFloorBounceDistanceRangeWU = new Vector2(0.08f, 0.15f);

    [Tooltip("Maksimi satunnais-käännön (yaw) haarukka asteina XZ-tasossa (per throw).")]
    [SerializeField] private Vector2 floorBounceRandomYawRangeDeg = new Vector2(0f, 25f);

    private float _throwFloorBounceDistWU;
    private float _throwMinFloorBounceDistWU;
    private float _throwRandomYawDeg;
    private bool _randomizedThisThrow;


    // ---------------------- Pomppujen määrä & matkan hallinta ----------------------
    [Header("Deflection: Bounce Count & Distance")]
    [Tooltip("Kuinka monta kimpoa enintään.")]
    [SerializeField] private int maxBounces = 2;

    // ---------------------- Ajoitus & nopeus ----------------------
    [Header("Deflection: Timing & Speed")]
    [Tooltip("Minimiaika yhdelle segmentille (tuntuma).")]
    [SerializeField] private float minSegmentDuration = 0.25f;

    [Tooltip("Maksimiaika yhdelle segmentille (tuntuma).")]
    [SerializeField] private float maxSegmentDuration = 0.55f;

    // ---------------------- Kaaren muoto pompun jälkeen ----------------------
    [Header("Deflection: Arc Shaping (Bounce Apex)")]
    [Tooltip("0..1: Pienennä pompun kaaren huippukorkeutta (apex).")]
    [SerializeField, Range(0f, 1f)] private float apexBounceScale = 0.6f;
```

```
    [Tooltip("Kova yläraja pompun apex-korkeudelle (WU).")]
    [SerializeField] private float maxBounceApexWU = 3.0f;

    [Tooltip("Apexin arvioinnissa käytettävä vähimmäisetäisyys, estää lyhyiden segmenttien ylikorkean kaaren.")]
    [SerializeField] private float apexEvalDistanceFloorWU = 4f;


    // ---------------------- Suunnan ohjaus & vakaus ----------------------
    [Tooltip("Sekunteina: estää välittömän uudelleenosuman samaan collideriin (ping-pong).")]
    [SerializeField] private float minBounceSeparation = 0.06f;

    [Tooltip("Pieni irrotus pinnasta uuden segmentin alussa, ehkäisee tarttumista.")]
    [SerializeField] private float pushOffWU = 0.02f;


    [Header("Floor Bounce (only)")]
    [SerializeField, Range(0.6f, 0.99f)]
    private float floorOnlyMinNormalY = 0.7f; // hyväksy pomppu vain lähes vaakasuorasta pinnasta

    [SerializeField, Range(0f, 25f)]
    private float floorBounceYawTowardGoalDeg = 6f; // käännä suuntaa kohti maalia tämän verran (asteina)

    private float _currentHorizSpeed; // segmenttikohtainen vaakanopeus


    [Header("Flight feel (time-warp around apex)")]
    [Tooltip("Pienin suhteellinen nopeus kaaren huipulla. 1 = ei hidastusta, 0.35 = selkeä hidastus.")]
    [SerializeField, Range(0.1f, 1f)] private float minSpeedScaleAtApex = 0.5f;

    // --- Flight feel (time-warp around apex) ---
    [Header("Flight feel (time-warp around apex)")]
    [Tooltip("Miten kapeasti hidastus kohdistuu huipun ympärille. Suurempi = kapeampi vyöhyke.")]
    [SerializeField, Range(0.5f, 5f)] private float apexSlowExponent = 2.0f;

    // --- Deflection timing/speed ---
    [Header("Deflection: Timing & Speed")]
    [Tooltip("< 1 = hidasta joka pompun jälkeen, 1 = pidä sama nopeus.")]
    [SerializeField, Range(0f, 1f)] private float bounceSpeedScale = 0.75f;

    [Tooltip("Ota käyttöön apex-hidastus kaaren aikana.")]
    [SerializeField] private bool useApexSlowdown = true;

    [Header("Distance-scaled speed")]
    [Tooltip("Skaalaa heiton vaakanopeus vaakasuoran etäisyyden (ruutujen) mukaan.")]
    [SerializeField] private bool useDistanceScaledSpeed = true;

    [Tooltip("Vaakanopeus 1 ruudun heitolla.")]
    [SerializeField] private float speedAtOneTile = 2f;

    [Tooltip("Kuinka paljon nopeus kasvaa per lisäruutu.")]
    [SerializeField] private float speedPerTile = 2f;
```

```
    [Tooltip("Nopeuden minimiklemmari (varmistaa ettei mene liian hitaaksi).")]
    [SerializeField] private float speedMinClamp = 1.5f;

    [Tooltip("Nopeuden maksimiklemmari (esim. max range ~14).")]
    [SerializeField] private float speedMaxClamp = 14f;

    [Tooltip("Ruutukoko WU:na. Jos WU == ruutu, pidä 1.")]
    [SerializeField] private float tileSizeWU = 1f;

    [Header("Audio")]
    [SerializeField] private AudioClip[] explosionSounds;
    [SerializeField] private float explosionVolume = 1f;
    [SerializeField] private float explosionMaxHearingDistance = 80f;
    [SerializeField] private AnimationCurve explosionVolumeRolloff = AnimationCurve.Linear(0, 1, 1, 0.2f);

    // sisäinen tila
    private int _bounces;
    private Collider _lastHitCollider;
    private float _lastHitTime;

    // ********************** Deterministic END ******************************

    [SyncVar] public uint actorUnitNetId;
    [SyncVar] public uint ownerPlayerNetId;
    [SyncVar] public int ownerTeamId = -1;

    public static event EventHandler OnAnyGranadeExploded;

    [SerializeField] private Transform grenadeExplodeVFXPrefab;
    [SerializeField] private float damageRadius = 4f;
    [SerializeField] private int damage = 30;

    [Header("Config")]
    [SerializeField] private ThrowArcConfig throwArcConfig;
    [SerializeField] private LayerMask ceilingMask;
    [SerializeField] private LayerMask floorMask;
    [SerializeField] private float ceilingClearance = 0.08f;
    [SerializeField] private float fallbackMaxThrowRangeWU = 12f;   // jos arvoa ei syötetä ulkoa
    [SerializeField] private float horizontalSpeed = 10f;           // tai käytä nykyistäsi


    [Header("Curve (fallback)")]
    [SerializeField] private AnimationCurve arcYAnimationCurve;

    private Vector3 _startPos;
    private Vector3 _endPos;
    private float _apexWU;          // dynaaminen huippukorkeus
    private float _travelT;         // 0..1
    private float _travelDuration;  // aika maaliin
    private float _maxRangeWU;

    // Pieni hajonta, muutaman sadasosan verran
```

```
    [SerializeField] private float explosionJitterMin = 0.02f;
    [SerializeField] private float explosionJitterMax = 0.08f;

    private bool _explosionScheduled; // vartija, ettei ajeta kahta kertaa

    [SyncVar(hook = nameof(OnTargetChanged))] private Vector3 targetPosition;

    private bool isExploded = false;

    // Synkattu räjähdyksen ajastus
    private bool  _armed;
    private double _explodeAt;    // server-aika jolloin räjähtää
    private bool _damageDone;    // server: damage tehty

    private GrenadeBeaconEffect beaconEffect;

    [Tooltip("Jos off niin räjähtää vihollisen vuoron lopussa")]
    [Header("Timer. How Meny actions Player can make before explosion")]
    [SerializeField] private bool actionBasedTimer = true;
    [SerializeField] private int timer = 2;
    private int turnBasedTimer = 2;

    void Awake()
    {
        var sc = GetComponent<SphereCollider>();
        if (sc && throwArcConfig)
            sc.radius = throwArcConfig.projectileRadiusWU;  // yksi totuus

        // jos haluat varmistaa fallback-käyrän
        if (!throwArcConfig && arcYAnimationCurve == null)
            arcYAnimationCurve = AnimationCurve.Linear(0, 0, 1, 0);

        beaconEffect = GetComponentInChildren<GrenadeBeaconEffect>();
        if (beaconEffect != null)
        {
            beaconEffect.SetTurnsUntilExplosion(timer);
        }
    }


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

    bool _subscribed;

    public void Setup(Vector3 targetWorld, float maxThrowRangeWU = -1f)
    {
        _maxRangeWU = (maxThrowRangeWU > 0f) ? maxThrowRangeWU : fallbackMaxThrowRangeWU;
        targetPosition = targetWorld;
        _explosionScheduled = false;
```

```
        isExploded = false;

        RecomputeDerived();

        // Tilaa tasan kerran, riippumatta AP/Turn -tilasta
        if (NetMode.ServerOrOff && !_subscribed)
        {
            _subscribed = true;
            if (actionBasedTimer)
            {
                Unit.ActionPointUsed += Unit_ActionPointsUsed;
                TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
            }
            else
            {
                TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
            }
        }

        if (!actionBasedTimer && GameModeManager.SelectedMode == GameMode.CoOp)
        {
            timer *= 2;
        }

        if (actionBasedTimer && GameModeManager.SelectedMode == GameMode.CoOp)
        {
            turnBasedTimer += 1;
        }
    }

    private void Unit_ActionPointsUsed(object sender, EventArgs e)
    {

        if (ownerTeamId == -1)
        {
            Debug.LogWarning("No owner");
            return;
        }
        if (_explosionScheduled || isExploded || ownerTeamId == TeamsID.CurrentTurnTeamId())
        {
            return;
        }

        timer -= 1;

        // VISUAALI: server käskee, clientit tekevät
        if (NetworkServer.active)
        {
            RpcBeaconTick();
        } else if (!NetworkClient.active)
        {
            // offline
```

```
            beaconEffect?.OnTurnAdvanced();
        }

        if (timer > 0) return;

        _explosionScheduled = true;

        if (NetworkServer.active)
        {
            RpcBeaconArmNow();                  // <<< TEE TÄMÄ
            ServerArmExplosion();

        } else if (!NetworkClient.active)
        {
            beaconEffect?.TriggerFinalCountdown();
            StartCoroutine(LocalExplodeAfterJitter());
        }
    }

    // --- TIMER → SCHEDULING ---
    private void TurnSystem_OnTurnChanged(object sender, EventArgs e)
    {
        if (_explosionScheduled || isExploded) return;

        turnBasedTimer -= 1;

        // VISUAALI
        if (NetworkServer.active)
        {
            RpcBeaconTick();                    // <<< TEE TÄMÄ
        }
        else if (!NetworkClient.active)
        {
            beaconEffect?.OnTurnAdvanced();
        }

        if (turnBasedTimer  > 0) return;

        _explosionScheduled = true;

        if (NetworkServer.active)
        {
            RpcBeaconArmNow();                  // <<< TEE TÄMÄ
            ServerArmExplosion();
        } else if (!NetworkClient.active)
        {
            beaconEffect?.TriggerFinalCountdown();
            StartCoroutine(LocalExplodeAfterJitter());
        }
    }

    private void OnDestroy()
```

```
    {
        if (!_subscribed) return;
        if (actionBasedTimer)
            Unit.ActionPointUsed -= Unit_ActionPointsUsed;
        else
            TurnSystem.Instance.OnTurnChanged -= TurnSystem_OnTurnChanged;
        _subscribed = false;
    }

    void OnTargetChanged(Vector3 _old, Vector3 _new)
    {
        // Kun SyncVar saapuu clientille, laske johdetut kentät sielläkin
        RecomputeDerived();
    }

    private void RecomputeDerived()
    {
        _startPos = transform.position;
        _endPos = targetPosition;

        // Per-heitto jitter offlineen
        if (!_randomizedThisThrow)
        {
            if (!NetMode.IsOnline)
            {
                _throwFloorBounceDistWU = UnityEngine.Random.Range(floorBounceDistanceRangeWU.x, floorBounceDistanceRangeWU.y);
                _throwMinFloorBounceDistWU = UnityEngine.Random.Range(minFloorBounceDistanceRangeWU.x, minFloorBounceDistanceRangeWU.y);
                _throwRandomYawDeg = UnityEngine.Random.Range(floorBounceRandomYawRangeDeg.x, floorBounceRandomYawRangeDeg.y);
            }
            else
            {
                _throwFloorBounceDistWU = _throwMinFloorBounceDistWU = _throwRandomYawDeg = 0f;
            }
            _randomizedThisThrow = true;
        }

        float dWU = Vector2.Distance(
            new Vector2(_startPos.x, _startPos.z),
            new Vector2(_endPos.x, _endPos.z));

        int samples = (throwArcConfig != null)
            ? Mathf.Clamp(throwArcConfig.EvaluateSegments(dWU, 2f), 12, 40)
            : Mathf.Clamp(12 + Mathf.RoundToInt(dWU / 2f) * 4, 12, 40);

        float farWU = (_maxRangeWU > 0f) ? _maxRangeWU
                                         : (throwArcConfig ? throwArcConfig.farRangeWU : dWU);

        _apexWU = ArcApexSolver.ComputeCeilingClampedApex(
            _startPos, _endPos, throwArcConfig, ceilingMask,
            ceilingClearance, samples, farWUOverride: farWU);

        // — nopeus & kesto etäisyyden mukaan —
```

```
        _currentHorizSpeed = useDistanceScaledSpeed
            ? ComputeDistanceScaledSpeed(dWU)
            : horizontalSpeed;

        _travelDuration = Mathf.Max(minSegmentDuration, dWU / Mathf.Max(0.01f, _currentHorizSpeed));
        _travelT = 0f;

        enabled = true;
    }

    private void Update()
    {
        if (_travelDuration <= 0f) return;

        // perusaskel
        float baseStep = Time.deltaTime / _travelDuration;
        float tNext = _travelT + baseStep;

        // (VALINNAINEN) kevyt huippufiilis kaarikäyrästä
        if (useApexSlowdown) {
            var curve = (throwArcConfig && throwArcConfig.arcYCurve != null) ? throwArcConfig.arcYCurve : arcYAnimationCurve;
            float peak = curve.Evaluate(_travelT);            // 0..1
            float k    = Mathf.Pow(peak, apexSlowExponent);   // korosta huippua
            float slow = Mathf.Lerp(1f, Mathf.Clamp(minSpeedScaleAtApex, 0.1f, 1f), k);
            tNext = _travelT + baseStep * slow;
        }

        _travelT = Mathf.Clamp01(tNext);
        float t = _travelT;

        // kaaren piste
        Vector3 prev = transform.position;
        Vector3 pos  = Vector3.Lerp(_startPos, _endPos, t);
        var useCurve = (throwArcConfig && throwArcConfig.arcYCurve != null) ? throwArcConfig.arcYCurve : arcYAnimationCurve;
        float baselineY = Mathf.Lerp(_startPos.y, _endPos.y, t);
        pos.y = baselineY + useCurve.Evaluate(t) * _apexWU;

        // kimpo
        if (useDeterministicDeflectionOffline) {
            if (TryDeterministicDeflectOffline(prev, pos)) return;
        }

        // siirrä & suuntaa
        Vector3 dir = pos - prev;
        transform.position = pos;
        if (dir.sqrMagnitude > 0.0001f) transform.forward = dir.normalized;

        if (t >= 1f)
        {
            Vector3 p = _endPos;
            float y = p.y;
```

```
            // Etsi KORKEIN osumapinta lattia+este -kerroksista
            // (älä ignooraa triggereitä, jos esteet ovat trigger-collidereita)
            var origin = p + Vector3.up * 3f;
            var hits = Physics.RaycastAll(origin, Vector3.down, 12f, floorMask, QueryTriggerInteraction.Collide);

            float highest = float.NegativeInfinity;
            for (int i = 0; i < hits.Length; i++) {
                // jos haluat varmistaa, ettei “seisota itseään”, ohita oma collider
                if (TryGetComponent<Collider>(out var selfCol) && hits[i].collider == selfCol)
                    continue;
                if (hits[i].point.y > highest) highest = hits[i].point.y;
            }
            if (highest > float.NegativeInfinity) y = highest;

            // Nosta kranaatti pintaan: oma puoli-korkeus + pieni epsilon, ettei klippaa
            float half = 0f;
            if (TryGetComponent<Collider>(out var col)) half = col.bounds.extents.y;
            y += half + 0.005f;

            // Aseta lopullinen laskeutumispaikka
            var landed = new Vector3(p.x, y, p.z);
            transform.position = landed;

            // Jos räjähdys/efekti käyttää _endPos:ia, pidä ne synkassa:
            _endPos = landed;

            enabled = false;
        }
    }

    private bool TryDeterministicDeflectOffline(Vector3 prevPos, Vector3 nextPos)
    {
        // 0) Pomppuraja
        if (_bounces >= maxBounces) return false;

        float stepDist = Vector3.Distance(prevPos, nextPos);
        if (stepDist <= 1e-6f) return false;

        // 1) SphereCast reitin suuntaan (KÄYTÄ deflectionMaskia, jossa lattia on mukana)
        float radius = 0.12f;
        if (TryGetComponent<SphereCollider>(out var sc)) radius = sc.radius;
        else if (throwArcConfig) radius = throwArcConfig.projectileRadiusWU;

        Vector3 stepDir = (nextPos - prevPos).normalized;

        if (!Physics.SphereCast(prevPos, radius, stepDir, out var hit, stepDist, deflectionMask, QueryTriggerInteraction.Ignore))
            return false;

        // 2) Estä ping-pong samaan collideriin heti perään
        if (hit.collider == _lastHitCollider && (Time.time - _lastHitTime) < minBounceSeparation)
            return false;
```

```
        // 3) Hyväksy VAIN lattia (normaalin Y iso)
        if (hit.normal.y < floorOnlyMinNormalY)
            return false;

        // 4) Heijastussuunta XZ-tasossa + deterministinen satunnainen kääntö
        Vector3 preDir = stepDir;
        Vector3 vRef   = Vector3.Reflect(preDir, hit.normal);

        vRef.y = 0f; preDir.y = 0f;
        if (vRef.sqrMagnitude < 1e-6f) vRef = preDir;
        vRef.Normalize(); preDir.Normalize();

        float randomYaw = (_throwRandomYawDeg > 0f)
            ? UnityEngine.Random.Range(-_throwRandomYawDeg, _throwRandomYawDeg)
            : 0f;
        vRef = Quaternion.AngleAxis(randomYaw, Vector3.up) * vRef;
        vRef.Normalize();

        // Käännä hieman kohti alkuperäistä maalia (aste-rajalla)
        Vector3 toGoal = _endPos - hit.point; toGoal.y = 0f;
        if (toGoal.sqrMagnitude > 1e-6f)
        {
            toGoal.Normalize();
            float delta = Vector3.SignedAngle(vRef, toGoal, Vector3.up);
            float yaw   = Mathf.Clamp(delta, -floorBounceYawTowardGoalDeg, floorBounceYawTowardGoalDeg);
            vRef = Quaternion.AngleAxis(yaw, Vector3.up) * vRef;
            vRef.Normalize();
        }

        // 5) Pituus: pysy ruudussa -> käytä per-heitto min/max-arvoa
        float remWU = Mathf.Max(_throwMinFloorBounceDistWU, _throwFloorBounceDistWU);

        // 6) Uusi päätepiste XZ:ssä
        Vector3 candidateEnd = hit.point + vRef * remWU;

        // Lattiakorkeus alas-raylla → oikea kerros
        float endY = candidateEnd.y;
        if (Physics.Raycast(candidateEnd + Vector3.up * 2f, Vector3.down, out var hitFloor, 10f, floorMask, QueryTriggerInteraction.Ignore))
            endY = hitFloor.point.y;
        candidateEnd.y = endY;

        // 7) Aseta uusi segmentti (pieni “irrotus” pinnasta)
        _startPos = hit.point + vRef * pushOffWU;
        _endPos   = candidateEnd;

        // 8) Uuden segmentin apex ja kesto (pidä kaari matalana pompussa)
        float dWU = Vector2.Distance(
            new Vector2(_startPos.x, _startPos.z),
            new Vector2(_endPos.x,   _endPos.z));

        int samples = (throwArcConfig != null)
            ? Mathf.Clamp(throwArcConfig.EvaluateSegments(dWU, 2f), 12, 40)
```

```
            : Mathf.Clamp(12 + Mathf.RoundToInt(dWU / 2f) * 4, 12, 40);

        float farWU = (_maxRangeWU > 0f) ? _maxRangeWU
                                         : (throwArcConfig != null ? throwArcConfig.farRangeWU : dWU);

        _apexWU = ArcApexSolver.ComputeCeilingClampedApex(
            _startPos, _endPos, throwArcConfig, ceilingMask,
            ceilingClearance, samples, farWUOverride: farWU
        );

        // (POISTETTU: ComputeArcPeakOut – arvoja ei käytetä tässä)

        // Hillitse pompun kaaren korkeutta
        float nominalFloorApex = (throwArcConfig != null)
            ? throwArcConfig.EvaluateApex(Mathf.Max(dWU, apexEvalDistanceFloorWU), farWU)
            : _apexWU;

        _apexWU  = Mathf.Min(_apexWU, nominalFloorApex);
        _apexWU *= Mathf.Clamp01(apexBounceScale);
        _apexWU  = Mathf.Min(_apexWU, maxBounceApexWU);

        // 9) Kesto & resetit
        // CHANGE: vaimenna nykyistä vaakanopeutta pompun jälkeen ja käytä sitä keston laskennassa
        _currentHorizSpeed = Mathf.Max(0.01f, _currentHorizSpeed * Mathf.Clamp01(bounceSpeedScale));

        float minDur = (_bounces == 0) ? minSegmentDuration : Mathf.Max(0.12f, 0.75f * minSegmentDuration);
        _travelDuration = Mathf.Clamp(dWU / _currentHorizSpeed, minDur, maxSegmentDuration);
        _travelT = 0f;

        // Visuaalisesti osumapisteestä eteenpäin
        transform.position = _startPos;
        transform.forward  = vRef;

        // 10) Kirjanpito
        _bounces++;
        _lastHitCollider = hit.collider;
        _lastHitTime     = Time.time;

        return true;
    }

    // Saattaa olla ihan hyödyllinen joskus tulevaisuudessa.
    private void ComputeArcPeakOut(out float apexWorldY, out float tPeak)
    {
        var curve = (throwArcConfig && throwArcConfig.arcYCurve != null)
            ? throwArcConfig.arcYCurve
            : arcYAnimationCurve;

        // Etsi kaaren maksimi arvo ja sitä vastaava t. (näytepohjainen, riittävän kevyt)
        float maxV = float.NegativeInfinity;
        float bestT = 0.5f;
        const int N = 20;
```

```
        for (int i = 0; i <= N; i++)
        {
            float t = i / (float)N;
            float v = curve.Evaluate(t);
            if (v > maxV) { maxV = v; bestT = t; }
        }

        // BaselineY huipulla + suhteellinen huippukorkeus * _apexWU
        float baselineY = Mathf.Lerp(_startPos.y, _endPos.y, bestT);
        apexWorldY = baselineY + maxV * _apexWU;
        tPeak = bestT;
    }

    private float ComputeDistanceScaledSpeed(float dWU)
    {
        // Muunna ruuduiksi (1 ruutu minimi, ettei 0-alue hyydytä)
        float tiles = dWU / Mathf.Max(1e-3f, tileSizeWU);
        tiles = Mathf.Max(1f, tiles);

        // Lineaarinen malli: v = v(1 tile) + perTile * (tiles - 1)
        float v = speedAtOneTile + speedPerTile * (tiles - 1f);

        // Pinni
        return Mathf.Clamp(v, speedMinClamp, speedMaxClamp);
    }

    // --- OFFLINE LOCAL EXPLOSION ---
    private IEnumerator LocalExplodeAfterJitter()
    {
        float delay = UnityEngine.Random.Range(explosionJitterMin, explosionJitterMax);
        yield return new WaitForSeconds(delay);
        LocalExplode();
    }

    /*
    private void LocalExplode()
    {
        if (isExploded) return;
        isExploded = true;

        // DAMAGE paikallisesti
        Collider[] hits = Physics.OverlapSphere(targetPosition, damageRadius);
        foreach (var c in hits)
        {
            if (c.TryGetComponent<Unit>(out var u))
                NetworkSync.ApplyDamageToUnit(u, damage, targetPosition, this.GetActorId());
            if (c.TryGetComponent<DestructibleObject>(out var d))
                NetworkSync.ApplyDamageToObject(d, damage, targetPosition, this.GetActorId());
        }

        // VFX paikallisesti
        OnAnyGranadeExploded?.Invoke(this, EventArgs.Empty);
```

```
        SpawnRouter.SpawnLocal(
            grenadeExplodeVFXPrefab.gameObject,
            targetPosition + Vector3.up * 1f,
            Quaternion.identity,
            source: transform
        );

        Destroy(gameObject);
    }
    */

    private void LocalExplode()
    {
        if (isExploded) return;
        isExploded = true;

        // DAMAGE paikallisesti
        Collider[] hits = Physics.OverlapSphere(targetPosition, damageRadius);
        foreach (var c in hits)
        {
            if (c.TryGetComponent<Unit>(out var u))
                NetworkSync.ApplyDamageToUnit(u, damage, targetPosition, this.GetActorId());
            if (c.TryGetComponent<DestructibleObject>(out var d))
                NetworkSync.ApplyDamageToObject(d, damage, targetPosition, this.GetActorId());
        }

        // AUDIO
        PlayExplosionSound(targetPosition);

        // VFX paikallisesti
        OnAnyGranadeExploded?.Invoke(this, EventArgs.Empty);
        SpawnRouter.SpawnLocal(
            grenadeExplodeVFXPrefab.gameObject,
            targetPosition + Vector3.up * 1f,
            Quaternion.identity,
            source: transform
        );

        Destroy(gameObject);
    }

    /*
        private IEnumerator SpawnVFXAt(string sceneName, Vector3 pos, double explodeAtServerTime)
        {
            float wait = Mathf.Max(0f, (float)(explodeAtServerTime - NetworkTime.time));
            if (wait > 0f) yield return new WaitForSeconds(wait);

            // AUDIO
            PlayExplosionSound(pos);

            // Vain visuaali – ei peli-impactia clienteillä
            OnAnyGranadeExploded?.Invoke(this, EventArgs.Empty);
```

```
            SpawnRouter.SpawnLocal(
                grenadeExplodeVFXPrefab.gameObject,
                pos + Vector3.up * 1f,
                Quaternion.identity,
                source: null,
                sceneName: sceneName
            );
        }

    */

    private IEnumerator SpawnVFXAt(string sceneName, Vector3 pos, double explodeAtServerTime)
    {
        float wait = Mathf.Max(0f, (float)(explodeAtServerTime - NetworkTime.time));
        if (wait > 0f) yield return new WaitForSeconds(wait);

        // AUDIO
        PlayExplosionSound(pos);

        // Vain visuaali — ei peli-impactia clienteillä
        OnAnyGranadeExploded?.Invoke(this, EventArgs.Empty);
        SpawnRouter.SpawnLocal(
            grenadeExplodeVFXPrefab.gameObject,
            pos + Vector3.up * 1f,
            Quaternion.identity,
            source: null,
            sceneName: sceneName
        );
    }

    private void PlayExplosionSound(Vector3 position)
    {
        if (explosionSounds == null || explosionSounds.Length == 0) return;

        AudioClip clip = explosionSounds[UnityEngine.Random.Range(0, explosionSounds.Length)];
        if (clip == null) return;

        GameObject audioGO = new GameObject("ExplosionAudio_Temp");
        audioGO.transform.position = position;

        AudioSource source = audioGO.AddComponent<AudioSource>();
        source.clip = clip;
        source.volume = explosionVolume;
        source.spatialBlend = 1f;
        source.rolloffMode = AudioRolloffMode.Custom;
        source.maxDistance = explosionMaxHearingDistance;
        source.minDistance = 5f;
        source.dopplerLevel = 0f;
        source.SetCustomCurve(AudioSourceCurveType.CustomRolloff, explosionVolumeRolloff);

        source.Play();
```

```
        Destroy(audioGO, clip.length + 0.1f);
    }



    [ClientRpc]
    private void RpcBeaconTick()
    {
        if (!beaconEffect) beaconEffect = GetComponentInChildren<GrenadeBeaconEffect>(true);
        beaconEffect?.OnTurnAdvanced();
    }

    [ClientRpc]
    private void RpcBeaconArmNow()
    {
        if (!beaconEffect) beaconEffect = GetComponentInChildren<GrenadeBeaconEffect>(true);
        beaconEffect?.TriggerFinalCountdown();
    }

    [Server]
    private void ServerArmExplosion()
    {
        // Jitter päälle (server päättää)
        float jitter = UnityEngine.Random.Range(explosionJitterMin, explosionJitterMax);
        _explodeAt = NetworkTime.time + jitter;

        // Kerro kaikille clienteille (hostin local client mukaan lukien)
        RpcArmExplosion(gameObject.scene.name, targetPosition, _explodeAt);

        // Server odottaa samaan serveriaikaan ja tekee vahingot
        StartCoroutine(ServerExplodeAt(_explodeAt));
    }

    [ClientRpc]
    private void RpcArmExplosion(string sceneName, Vector3 pos, double explodeAtServerTime)
    {
        if (_armed) return; // varmistus
        _armed = true;
        _explodeAt = explodeAtServerTime;

        // Ajasta paikallinen VFX täsmälleen samaan aikaan serverin kellossa
        StartCoroutine(SpawnVFXAt(sceneName, pos, explodeAtServerTime));
    }

    [Server]
    private IEnumerator ServerExplodeAt(double explodeAtServerTime)
    {
        // Odota tarkkaan serverkellon mukaan
        float wait = Mathf.Max(0f, (float)(explodeAtServerTime - NetworkTime.time));
        if (wait > 0f) yield return new WaitForSeconds(wait);

        if (_damageDone) yield break;
```

```
        _damageDone = true;
        if (isExploded) yield break;
        isExploded = true;

        // DAMAGE vain serverillä (autoritäärinen peli-impact)
        Collider[] hits = Physics.OverlapSphere(targetPosition, damageRadius);
        foreach (var c in hits)
        {
            if (c.TryGetComponent<Unit>(out var u))
                NetworkSync.ApplyDamageToUnit(u, damage, targetPosition, this.GetActorId());
            if (c.TryGetComponent<DestructibleObject>(out var d))
                NetworkSync.ApplyDamageToObject(d, damage, targetPosition, this.GetActorId());
        }

        // Piilota itse kranaatti kaikilta
        SetSoftHiddenLocal(true);
        RpcSetSoftHidden(true);

        // Server voi nyt siivota objektin pienen viiveen jälkeen
        StartCoroutine(DestroyAfter(0.30f));
    }


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

        if (NetworkServer.active)    // online-server
            NetworkServer.Destroy(gameObject);
        else                         // offline
            Destroy(gameObject);
    }

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

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

}
```

Assets/scripts/Combat/Weapons/Grenade/GrenadeArcPreview.cs

```
using UnityEngine;

[RequireComponent(typeof(LineRenderer))]
public class GrenadeArcPreview : MonoBehaviour
{
    [SerializeField] private ThrowArcConfig throwArcConfig;
    [SerializeField] private AnimationCurve arcYAnimationCurve; // käytetään jos config puuttuu
    [SerializeField] private Transform origin;                  // tyypillisesti UnitAnimator.ThrowPoint
   // [SerializeField] private float fallbackMaxThrowRangeWU = 12f;
    [SerializeField] private float cellSizeWU = 2f;

    // Katon tunnistus esikatselulle
    [SerializeField] private LayerMask ceilingMask;      // VAIN “Ceiling”-layer
    [SerializeField] private bool clampToCeiling = true; // ON = käytä kakkoskaarta sisällä
    [SerializeField] private float ceilingClearance = 0.08f;

    // Segmenttien rajat
    [SerializeField] private int minSegments = 12;
    [SerializeField] private int maxSegments = 40;

    private LineRenderer _lr;

    void Awake()
    {
        _lr = GetComponent<LineRenderer>();
        _lr.enabled = false;
        if (throwArcConfig != null && throwArcConfig.arcYCurve != null)
            arcYAnimationCurve = throwArcConfig.arcYCurve;
    }

    // Säilytä vanha signatuuri – se delegoi uuteen overloadiin
    public void ShowArcTo(Vector3 targetWorldPos, float maxThrowRangeWU = -1f)
    {
        ShowArcTo(targetWorldPos, null, maxThrowRangeWU);
    }

    // UUSI: mahdollistaa myös apex-override’n (jos joskus haluat syöttää sen suoraan)
    public void ShowArcTo(Vector3 targetWorldPos, float? apexOverrideWU, float maxThrowRangeWU = -1f)
    {
        if (origin == null || _lr == null)
        {
            Debug.LogWarning("[GrenadeArcPreview] Ei originia tai LineRendereriä.");
            return;
        }

        _lr.enabled = true;

        Vector3 start = origin.position;
        Vector3 end = targetWorldPos;
```

```
        // Vaakasuora etäisyys
        Vector2 s = new Vector2(start.x, start.z);
        Vector2 e = new Vector2(end.x, end.z);
        float dWU = Vector2.Distance(s, e);

        // Käytä KÄYTÄNNÖSSÄ annettua maxThrowRangeWU:ta, jos se on > 0
        float farWU = (maxThrowRangeWU > 0f) ? maxThrowRangeWU
                                             : (throwArcConfig != null ? throwArcConfig.farRangeWU : dWU);

        // Segmenttien määrä
        int segs = (throwArcConfig != null)
            ? Mathf.Clamp(throwArcConfig.EvaluateSegments(dWU, Mathf.Max(0.01f, cellSizeWU)), minSegments, maxSegments)
            : Mathf.Clamp(12 + Mathf.RoundToInt(dWU / Mathf.Max(0.01f, cellSizeWU)) * 4, minSegments, maxSegments);

        // Apeksin valinta: 1) suora override 2) katto-clamp 3) normaali
        float apexWU;
        if (apexOverrideWU.HasValue)
        {
            apexWU = Mathf.Max(0f, apexOverrideWU.Value);
        }
        else if (clampToCeiling && ceilingMask.value != 0)
        {
            // Kattoa vasten clampattu “kakkoskaari”
            apexWU = ArcApexSolver.ComputeCeilingClampedApex(
                start, end, throwArcConfig, ceilingMask,
                ceilingClearance: ceilingClearance, samples: segs
            );
        }
        else
        {
            // Normaali kaari
            apexWU = (throwArcConfig != null)
                ? throwArcConfig.EvaluateApex(dWU, farWU) // HUOM: käytä farWU’tä (aiemmin jäi hyödyntämättä)
                : Mathf.Lerp(7f, 1.2f, Mathf.Clamp01(dWU / 12f));
        }

        // Piirrä kaari
        _lr.positionCount = segs + 1;
        var curve = (throwArcConfig && throwArcConfig.arcYCurve != null)
            ? throwArcConfig.arcYCurve
            : arcYAnimationCurve;

        for (int i = 0; i <= segs; i++)
        {
            float t = i / (float)segs;

            Vector3 p = Vector3.Lerp(start, end, t);
            float baselineY = Mathf.Lerp(start.y, end.y, t);
            float yArc = curve.Evaluate(t) * apexWU;

            p.y = baselineY + yArc;
            _lr.SetPosition(i, p);
```

```
        }
    }

    public void Hide() => _lr.enabled = false;

    // Aseta nämä dynaamisesti Actionista tarvittaessa:
    public void SetOrigin(Transform t) => origin = t;
    public void SetCellSize(float size) => cellSizeWU = size;
}
```

Assets/scripts/Combat/Weapons/Grenade/GrenadeBeaconEffect.cs

```
using UnityEngine;

[RequireComponent(typeof(Light))]
public class GrenadeBeaconEffect : MonoBehaviour
{
    [Header("Visual Settings")]
    [SerializeField] private Color beaconColor = Color.red;
    [SerializeField] private float minIntensity = 0.5f;
    [SerializeField] private float maxIntensity = 3f;
    [SerializeField] private float minRange = 2f;
    [SerializeField] private float maxRange = 5f;
    [SerializeField] private float basePulseSpeed = 3f;

    [Header("Turn-Based Timing")]
    [SerializeField] private int turnsUntilExplosion = 2;
    [SerializeField] private float finalPulseSpeedMultiplier = 5f;

    [Header("Audio Settings")]
    [SerializeField] private AudioSource audioSource;
    [SerializeField] private AudioClip beepSound;
    [SerializeField] private float basePitch = 1f;
    [SerializeField] private float finalPitchMultiplier = 1.5f;

    private Light beaconLight;
    private int currentTurnsRemaining;
    private bool isArmed = false;
    private float currentPulseSpeed;
    private float currentPitch;

    void Awake()
    {
        beaconLight = GetComponent<Light>();

        if (beaconLight == null)
        {
            beaconLight = gameObject.AddComponent<Light>();
        }

        beaconLight.type = LightType.Point;
        beaconLight.color = beaconColor;
        beaconLight.intensity = minIntensity;
        beaconLight.range = minRange;
        beaconLight.renderMode = LightRenderMode.ForcePixel;

        if (audioSource == null)
        {
            audioSource = GetComponent<AudioSource>();
            if (audioSource == null)
            {
                audioSource = gameObject.AddComponent<AudioSource>();
```

```
            audioSource.playOnAwake = false;
            audioSource.spatialBlend = 1f;
            audioSource.minDistance = 3f;
            audioSource.maxDistance = 20f;
        }
    }

    currentTurnsRemaining = turnsUntilExplosion;
    currentPulseSpeed = basePulseSpeed;
    currentPitch = basePitch;
}

void Update()
{
    UpdateLightPulse();
}

public void OnTurnAdvanced()
{
    if (currentTurnsRemaining > 0)
    {
        currentTurnsRemaining--;
        UpdatePulseParameters();
        PlayBeep();
    }

    if (currentTurnsRemaining == 0)
    {
        isArmed = true;
    }
}

public void TriggerFinalCountdown()
{
    isArmed = true;
    currentTurnsRemaining = 0;
    currentPulseSpeed = basePulseSpeed * finalPulseSpeedMultiplier;
    currentPitch = basePitch * finalPitchMultiplier;
}

private void UpdatePulseParameters()
{
    if (turnsUntilExplosion <= 0) return;

    float progress = 1f - ((float)currentTurnsRemaining / turnsUntilExplosion);

    currentPulseSpeed = Mathf.Lerp(basePulseSpeed, basePulseSpeed * finalPulseSpeedMultiplier, progress);
    currentPitch = Mathf.Lerp(basePitch, basePitch * finalPitchMultiplier, progress);
}

private void UpdateLightPulse()
{
```

```
        float pulseValue = (Mathf.Sin(Time.time * currentPulseSpeed) + 1f) * 0.5f;

        beaconLight.intensity = Mathf.Lerp(minIntensity, maxIntensity, pulseValue);
        beaconLight.range = Mathf.Lerp(minRange, maxRange, pulseValue);
    }

    private void PlayBeep()
    {
        if (audioSource != null && beepSound != null)
        {
            audioSource.pitch = currentPitch;
            audioSource.PlayOneShot(beepSound);
        }
    }

    public void SetTurnsUntilExplosion(int turns)
    {
        turnsUntilExplosion = turns;
        currentTurnsRemaining = turns;
        UpdatePulseParameters();
    }

    public bool IsArmed => isArmed;
    public int TurnsRemaining => currentTurnsRemaining;

    public void SetRemainingDirect(int remaining)
    {
        currentTurnsRemaining = Mathf.Max(remaining, 0);
        UpdatePulseParameters();
    }

    // Soita yksi beep (RPC:tä varten)
    public void PlayBeepOnce()
    {
        PlayBeep();
    }
}
```

Assets/scripts/Combat/Weapons/Grenade/GrenadeDefinition.cs

```
using UnityEngine;

public enum EffectArea { Melee, Close, Medium }
public enum ThrowTier { CritMiss, Miss, Hit, Bullseye }
public enum GrenadeType  {Frag, flash, Smoke}

[CreateAssetMenu(menuName = "RogueShooter/Grenade")]
public class GrenadeDefinition : ScriptableObject
{
    [Header("Base damage")]
    public int baseDamage = 300;
    public float pressureFactor = 0.2f; // Effects only Unit cover skill and light covers and objects.

    [Header("Base hit chance baseline by band (% before skill) Overall must be 100%")]
    public int critMiss = 10;
    public int miss = 20;
    public int hit = 60;
    public int Bullseye = 10;

    [Header("Timer(turns before explotion)")]
    public int timer = 1;
}
```

Assets/scripts/Combat/Weapons/Grenade/ThrowArcConfig.cs

```
using UnityEngine;

[CreateAssetMenu(menuName = "RogueShooter/Throw Arc Config")]
public class ThrowArcConfig : ScriptableObject
{
    [Header("Arc shape (0→1→0, huippu ~0.5)")]
    public AnimationCurve arcYCurve = new AnimationCurve(
        new Keyframe(0f, 0f, 0f, 4f),
        new Keyframe(0.5f, 1f, 0f, 0f),
        new Keyframe(1f, 0f, -4f, 0f)
    );

    [Header("Apex (world-yksiköissä)")]
    [Tooltip("Kuinka korkea kaaren huippu on lyhyellä heitolla.")]
    public float apexNearWU = 7f;
    [Tooltip("Kuinka matala kaaren huippu on maksietäisyydellä.")]
    public float apexFarWU  = 1.2f;

    [Header("Smoothing")]
    [Tooltip("0.5-1.0: pienempi → voimakkaampi ero near vs far")]
    [Range(0.25f, 2f)] public float smoothingK = 0.75f;

    [Header("Etäisyysskaala")]
    [Tooltip("Maksimi heittoetäisyys world-yksiköissä (esim. throwingRangeTiles * cellSizeWU).")]
    public float farRangeWU = 12f;

    [Header("Landing Angle Gate")]
    [Tooltip("Minimilaskukulma (asteina) kun ruudussa on obstacle. Esim. 35–55°.")]
    public float minLandingAngleDegForObstacles = 40f;

    [Tooltip("Näytteenottokohdat kulman arvioon (lähellä laskeutumista).")]
    [Range(0f,1f)] public float angleSampleT1 = 0.92f;
    [Range(0f,1f)] public float angleSampleT2 = 0.98f;

    [Header("Preview")]
    public int baseSegments = 12;
    public int segmentsPerTile = 4;
    public int minSegments = 12;
    public int maxSegments = 40;

    [Header("Projectile / Collision")]
    [Tooltip("Kranaatin todellinen säde (WU). Käytetään sekä fyysisessä colliderissa että arc-tarkistuksessa.")]
    public float projectileRadiusWU = 0.12f;

    [Tooltip("Vähimmäissuhde ruudun kokoon, jolla kapseli 'ottaa kiinni' seinäkulmiin.")]
    [Range(0f, 0.5f)] public float capsuleRadiusMinRelToCell = 0.20f;

    // Yhtenäinen tapa hakea tarkistuksissa käytettävä kapselisäde
    public float GetCapsuleRadius(float cellSizeWU)
        => Mathf.Max(projectileRadiusWU, cellSizeWU * capsuleRadiusMinRelToCell);
```

```
    public float EvaluateApex(float distanceWU, float farRangeWU)
    {
        float x = farRangeWU > 0f ? Mathf.Clamp01(distanceWU / farRangeWU) : 1f;
        // "Käänteinen" smoothstep: lähellä korkea, kaukana matala
        float s = 1f - Mathf.Pow(1f - x, smoothingK);
        return Mathf.Lerp(apexNearWU, apexFarWU, s);
    }

    public int EvaluateSegments(float distanceWU, float cellSizeWU)
    {
        int tiles = Mathf.Max(0, Mathf.RoundToInt(distanceWU / Mathf.Max(0.01f, cellSizeWU)));
        int segs = baseSegments + tiles * segmentsPerTile;
        return Mathf.Clamp(segs, minSegments, maxSegments);
    }
}
```

Assets/scripts/Combat/Weapons/Gun/BulletProjectile.cs

```
using Mirror;
using UnityEngine;

public class BulletProjectile : NetworkBehaviour
{
    [SyncVar] public uint actorUnitNetId;

    [Header("Visual")]
    [SerializeField] private TrailRenderer trailRenderer;

    [Header("Impact VFX")]
    [SerializeField] private Transform unitHitVfxPrefab;
    [SerializeField] private Transform environmentHitVfxPrefab;

    [Header("Settings")]
    [SerializeField] private float moveSpeed = 200f;
    [SerializeField] private float bulletRadius = 0.05f;
    [SerializeField] private float maxLifetime = 5f;
    [SerializeField] private float maxTravelDistance = 100f;

    [SyncVar] private Vector3 targetPosition;
    [SyncVar] private bool shouldHitUnits;

    private float spawnTime;
    private Vector3 startPosition;
    private bool hasHit;
    private Vector3 flyDirection;

    private LayerMask unitsLayerMask;
    private LayerMask environmentLayerMask;

    public void Setup(Vector3 targetPosition, bool canHitUnits)
    {
        this.targetPosition = targetPosition;
        this.shouldHitUnits = canHitUnits;
    }

    public override void OnStartClient()
    {
        base.OnStartClient();

        if (trailRenderer && !trailRenderer.emitting)
            trailRenderer.emitting = true;

        spawnTime = Time.time;
        startPosition = transform.position;
        flyDirection = (targetPosition - startPosition).normalized; // ← LASKE KERRAN ALUSSA

        SetupLayerMasks();
    }
```

```
   private void Start()
    {
        spawnTime = Time.time;
        startPosition = transform.position;
        flyDirection = (targetPosition - startPosition).normalized; // ← LASKE KERRAN ALUSSA

        SetupLayerMasks();
    }

    private void SetupLayerMasks()
    {
        int unitsLayer = LayerMask.NameToLayer("Units");
        int obstaclesLayer = LayerMask.NameToLayer("Obstacles");
        int narrowObstaclesLayer = LayerMask.NameToLayer("NarrowObstacles"); // ← KORJAA NIMI (poista "Layer")
        int defaultLayer = LayerMask.NameToLayer("Default");
        int floorLayer = LayerMask.NameToLayer("Floor");

        unitsLayerMask = 1 << unitsLayer;
        environmentLayerMask = (1 << obstaclesLayer) | (1 << narrowObstaclesLayer) | (1 << defaultLayer) | (1 << floorLayer);

//        Debug.Log($"[BULLET SETUP] Units: {unitsLayer}, Env layers: Obstacles({obstaclesLayer}), NarrowObstacles({narrowObstaclesLayer}), Default({defaultLayer}),
Floor({floorLayer})");
    }

    private void Update()
    {
        if (hasHit) return;

        if (Time.time - spawnTime > maxLifetime)
        {
       //     Debug.Log($"[BULLET] Tuhottu: maxLifetime");
            DestroyBulletSilently();
            return;
        }

        float traveledDistance = Vector3.Distance(startPosition, transform.position);
        if (traveledDistance > maxTravelDistance)
        {
       //     Debug.Log($"[BULLET] Tuhottu: maxTravelDistance");
            DestroyBulletSilently();
            return;
        }

        Vector3 startPos = transform.position;
        float moveDistance = moveSpeed * Time.deltaTime;

        // ← KÄYTÄ flyDirection, ÄLÄ laske uudelleen!

        // 1. Tarkista ympäristö
        if (Physics.SphereCast(
            startPos,
```

```
            bulletRadius,
            flyDirection, // ← KÄYTÄ TALLENNETTUA SUUNTAA
            out RaycastHit envHit,
            moveDistance,
            environmentLayerMask,
            QueryTriggerInteraction.Ignore))
        {
        //    Debug.Log($"[BULLET] OSUI YMPÄRISTÖÖN: {envHit.collider.name}, Layer: {LayerMask.LayerToName(envHit.collider.gameObject.layer)}");
            HandleHit(envHit, false);
            return;
        }

        // 2. Tarkista Unitit
        if (shouldHitUnits)
        {
            if (Physics.SphereCast(
                startPos,
                bulletRadius,
                flyDirection, // ← KÄYTÄ TALLENNETTUA SUUNTAA
                out RaycastHit unitHit,
                moveDistance,
                unitsLayerMask,
                QueryTriggerInteraction.Ignore))
            {
          //      Debug.Log($"[BULLET] OSUI UNITTIIN: {unitHit.collider.name}");
                HandleHit(unitHit, true);
                return;
            }
        }

        // 3. Liikuta eteenpäin SAMAAN SUUNTAAN
        transform.position += flyDirection * moveDistance;
    }



    private void HandleHit(RaycastHit hit, bool isUnit)
    {
        if (hasHit) return;
        hasHit = true;

        Vector3 hitPoint = hit.point;
        Vector3 hitNormal = hit.normal;

        Transform vfxPrefab = isUnit ? unitHitVfxPrefab : environmentHitVfxPrefab;

        if (vfxPrefab != null)
        {
            Quaternion rotation = Quaternion.LookRotation(hitNormal);

            SpawnRouter.SpawnLocal(
```

```
                vfxPrefab.gameObject,
                hitPoint,
                rotation,
                source: transform
            );
        }

        DestroyBullet(hitPoint);
    }

    private void DestroyBullet(Vector3 finalPosition)
    {
        transform.position = finalPosition;

        if (trailRenderer)
        {
            trailRenderer.transform.SetParent(null);
            trailRenderer.emitting = false;
            Destroy(trailRenderer.gameObject, trailRenderer.time);
        }

        if (isServer)
            NetworkServer.Destroy(gameObject);
        else
            Destroy(gameObject);
    }

    private void DestroyBulletSilently()
    {
        if (trailRenderer)
        {
            trailRenderer.transform.SetParent(null);
            trailRenderer.emitting = false;
            Destroy(trailRenderer.gameObject, trailRenderer.time);
        }

        if (isServer)
            NetworkServer.Destroy(gameObject);
        else
            Destroy(gameObject);
    }
}
```

Assets/scripts/Combat/Weapons/Gun/OverwatchVisionUpdater.cs

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

public static class OverwatchVisionUpdater
{
    // --- lisätty: kevyt kuristus online-lähetyksille ---
    private const float SEND_INTERVAL = 0.1f;  // ~5 Hz
    private const float MIN_DEG_STEP = 1f;     // lähetä vasta kun kääntynyt ≥ 6°
    private static readonly Dictionary<int, float> s_nextSendAt = new();
    private static readonly Dictionary<int, byte> s_lastYawQ = new();

    public static void UpdateVision(Unit unit, Vector3 facingWorld, float coneAngle)
    {
        if (unit == null) return;

        facingWorld = OverwatchHelpers.NormalizeFacing(facingWorld);

        UpdatePayload(unit, facingWorld, coneAngle);
        UpdateLocalOverlay(unit, facingWorld, coneAngle);
        UpdateVisionCacheAndTeamVision(unit, facingWorld, coneAngle);
    }

    private static void UpdatePayload(Unit unit, Vector3 facingWorld, float coneAngle)
    {
        if (unit.TryGetComponent<UnitStatusController>(out var status))
        {
            if (status.TryGet<OverwatchPayload>(UnitStatusType.Overwatch, out var payload))
            {
                payload.facingWorld = facingWorld;
                payload.coneAngleDeg = coneAngle;
                status.AddOrUpdate(UnitStatusType.Overwatch, payload);
            }
        }
    }

    private static void UpdateLocalOverlay(Unit unit, Vector3 facingWorld, float coneAngle)
    {
        if (unit.TryGetComponent<UnitVision>(out var vision))
        {
            vision.ShowUnitOverWatchVision(facingWorld, coneAngle);
        }
    }

    private static void UpdateVisionCacheAndTeamVision(Unit unit, Vector3 facingWorld, float coneAngle)
    {
        if (NetworkSync.IsOffline)
        {
            UpdateVisionOffline(unit, facingWorld, coneAngle);
        }
```

```
        else if (Mirror.NetworkServer.active && NetworkSyncAgent.Local != null)
        {
            // --- lisätty: kuristus ennen raskasta polkua ---
            if (!ShouldSendOnline(unit, facingWorld)) return;

            UpdateVisionOnline(unit, facingWorld, coneAngle);
        }
    }

    private static void UpdateVisionOffline(Unit unit, Vector3 facingWorld, float coneAngle)
    {
        if (unit.TryGetComponent<UnitVision>(out var vision) && vision.IsInitialized)
        {
            vision.UpdateVisionNow();
            vision.ApplyAndPublishDirectionalVision(facingWorld, coneAngle);
        }
    }

    private static void UpdateVisionOnline(Unit unit, Vector3 facingWorld, float coneAngle)
    {
        if (unit.TryGetComponent<UnitVision>(out var vision) && vision.IsInitialized)
        {
            vision.UpdateVisionNow();
            vision.ApplyAndPublishDirectionalVision(facingWorld, coneAngle);
        }

        var ni = unit.GetComponent<Mirror.NetworkIdentity>();
        if (ni != null)
        {
            NetworkSyncAgent.Local.RpcUpdateSingleUnitVision(ni.netId, facingWorld, coneAngle);
        }
    }

    // --- lisätty: yksinkertainen kulma- ja aika-kuristin ---
    private static bool ShouldSendOnline(Unit unit, Vector3 facingWorld)
    {
        int id = unit.GetInstanceID();
        float now = Time.time;

        if (s_nextSendAt.TryGetValue(id, out var next) && now < next)
            return false;

        // kvantisoi yaw 0..255 (halpa tapa mitata kulmamuutos)
        float yaw360 = Mathf.Repeat(Mathf.Atan2(facingWorld.x, facingWorld.z) * Mathf.Rad2Deg, 360f);
        byte yawQ = (byte)Mathf.RoundToInt(yaw360 * 255f / 360f);

        if (s_lastYawQ.TryGetValue(id, out var last))
        {
            float degStep = Mathf.Abs((yawQ - last) * (360f / 255f));
            if (degStep < MIN_DEG_STEP) return false;
        }
```

```
        s_lastYawQ[id] = yawQ;
        s_nextSendAt[id] = now + SEND_INTERVAL;
        return true;
    }
}
```

Assets/scripts/Combat/Weapons/Gun/WeaponDefinition.cs

```
using UnityEngine;

public enum RangeBand { Melee, Close, Medium, Long, Extreme }
public enum ShotTier  { CritMiss, Close, Graze, Hit, Crit }

[System.Serializable]
public struct RangeBandTuning
{
    [Header("Stage 1: Base chance to HIT (before skill/cover)")]
    [Range(0, 100)] public int baseHitChance;

    [Header("Stage 2: On HIT distribution (Close/Graze/Hit/Crit)")]
    [Range(0, 100)] public int onHit_Close;
    [Range(0, 100)] public int onHit_Graze;
    [Range(0, 100)] public int onHit_Hit;
    [Range(0, 100)] public int onHit_Crit;
}

[System.Serializable]
public struct NormalShootingSettings
{
    [Header("Shooting Tempo")]
    [Tooltip("Kääntymisen nopeus normaalissa ampumisessa (deg/s)")]
    [Range(10f, 600f)] public float turnSpeed;

    [Tooltip("Minimiaika tähtäämiseen")]
    [Range(0.05f, 1.0f)] public float minAimTime;

    [Tooltip("Kokonaisaika tähtäystilassa")]
    [Range(0.1f, 2.0f)] public float aimingStateTime;

    [Tooltip("Aika ampumistilan jälkeen")]
    [Range(0.1f, 2.0f)] public float cooloffStateTime;

    public static NormalShootingSettings Default => new NormalShootingSettings
    {
        turnSpeed = 45f,
        minAimTime = 0.40f,
        aimingStateTime = 1.00f,
        cooloffStateTime = 0.50f
    };
}

[System.Serializable]
public struct OverwatchShootingSettings
{
    [Header("Geometry")]
    [Tooltip("Overwatch-kartion kulma asteina")]
    [Range(30f, 180f)] public float coneAngleDeg;
```

```
    [Tooltip("Overwatchin kantama ruutuina")]
    [Range(1, 15)] public int rangeTiles;

    [Header("Shooting Tempo")]
    [Tooltip("Kääntymisen nopeus overwatch-laukauksissa (deg/s)")]
    [Range(10f, 600f)] public float turnSpeed;

    [Tooltip("Minimiaika tähtäämiseen")]
    [Range(0.05f, 0.5f)] public float minAimTime;

    [Tooltip("Kokonaisaika tähtäystilassa")]
    [Range(0.1f, 1.0f)] public float aimingStateTime;

    [Tooltip("Aika ampumistilan jälkeen")]
    [Range(0.1f, 1.0f)] public float cooloffStateTime;

    [Header("Reaction Timing")]
    [Tooltip("Satunnainen viive reaktioon (maksimi)")]
    [Range(0f, 0.25f)] public float reactionJitterMaxSeconds;

    [Tooltip("Minimi-aikaväli kahden overwatch-reaktion välillä")]
    [Range(0.1f, 2.0f)] public float reactionCooldownSeconds;

    [Tooltip("Overwatch shooting accuracy penalty")]
    [Range(0f, 100f)] public float overwatchShootPenalty;

    public static OverwatchShootingSettings Default => new OverwatchShootingSettings
    {
        coneAngleDeg = 80f,
        rangeTiles = 8,
        turnSpeed = 45f,
        minAimTime = 0.15f,
        aimingStateTime = 0.5f,
        cooloffStateTime = 0.3f,
        reactionJitterMaxSeconds = 0.10f,
        reactionCooldownSeconds = 0.5f,
        overwatchShootPenalty = 10f
    };
}

[CreateAssetMenu(menuName = "RogueShooter/Weapon")]
public class WeaponDefinition : ScriptableObject
{
    [Header("Normal Shooting")]
    [Tooltip("Normaali ampuminen (deg/s + vaiheet)")]
    public NormalShootingSettings normalShooting = NormalShootingSettings.Default;

    [Header("Overwatch Mode")]
    public OverwatchShootingSettings overwatch = OverwatchShootingSettings.Default;

    [Header("Range: Weapon basic max Range (No upgrades)")]
    public int maxShootRange = 10;
```

```
    [Header("Bonus when the target is not behind cover")]
    public int NoCoverDamageBonus = 30;

    [Header("Base damage")]
    public int baseDamage = 10;
    public int critBonusDamage = 8;
    public float grazeFactor = 0.4f;
    public float missChipFactor = 0.2f;

    [Header("Legacy per-weapon ranges (used if no global CombatRanges found)")]
    public float closeMax = 4f;
    public float mediumMax = 9f;
    public float longMax = 15f;

    [Header("Advanced accuracy: per-band tunables")]
    public bool useAdvancedAccuracy = true;
    public RangeBandTuning melee;
    public RangeBandTuning close;
    public RangeBandTuning medium;
    public RangeBandTuning @long;
    public RangeBandTuning extreme;

    public RangeBandTuning GetTuning(RangeBand b)
    {
        switch (b)
        {
            case RangeBand.Melee:  return melee;
            case RangeBand.Close:  return close;
            case RangeBand.Medium: return medium;
            case RangeBand.Long:   return @long;
            default:               return extreme;
        }
    }

    [Header("Burst Fire Settings")]
    [Tooltip("Minimum shots per burst (e.g., 3 for assault rifles)")]
    public int burstMin = 3;
    [Tooltip("Maximum shots per burst (e.g., 4 for assault rifles)")]
    public int burstMax = 4;
    [Tooltip("Time between shots in a burst (seconds)")]
    public float burstShotDelay = 0.1f;

    public int GetRandomBurstCount() => Random.Range(burstMin, burstMax + 1);

    [Header("Legacy baselines (ignored if useAdvancedAccuracy==true)")]
    public int meleeAcc   = 95;
    public int closeAcc   = 80;
    public int mediumAcc  = 65;
    public int longAcc    = 45;
    public int extremeAcc = 25;
```

```
    [Header("Legacy crit starts (ignored if useAdvancedAccuracy==true)")]
    public int critStartMelee   = 90;
    public int critStartClose   = 85;
    public int critStartMedium  = 80;
    public int critStartLong    = 70;
    public int critStartExtreme = 60;

#if UNITY_EDITOR
    private void OnValidate()
    {
        if (normalShooting.turnSpeed < 5f)
            normalShooting.turnSpeed = 45f;

        if (overwatch.turnSpeed < 5f)
            overwatch.turnSpeed = Mathf.Max(normalShooting.turnSpeed, 20f);

        if (normalShooting.minAimTime <= 0f)       normalShooting.minAimTime       = 0.40f;
        if (normalShooting.aimingStateTime <= 0f)  normalShooting.aimingStateTime  = 1.00f;
        if (normalShooting.cooloffStateTime <= 0f) normalShooting.cooloffStateTime = 0.50f;

        if (overwatch.minAimTime <= 0f)       overwatch.minAimTime       = 0.15f;
        if (overwatch.aimingStateTime <= 0f)  overwatch.aimingStateTime  = 0.50f;
        if (overwatch.cooloffStateTime <= 0f) overwatch.cooloffStateTime = 0.30f;
    }
#endif
}
```

Assets/scripts/CoverSystem/CoverService.cs

```
public static class CoverService
{
    public enum CoverType { None, Low, High }

    public static float GetCoverMitigationBase(CoverType t)
        => t == CoverType.High ? .4f : (t == CoverType.Low ? .6f : 0);

    public static float GetCoverMitigationPoints(CoverType t)
    {
        float basePts = GetCoverMitigationBase(t);
        return basePts;
    }

    public static CoverType EvaluateCoverHalfPlane(GridPosition attacker, GridPosition target, PathNode node)
    {

        if (attacker.floor != target.floor) return CoverType.None; // pidä yksinkertaisena

        int dx = attacker.x - target.x;
        int dz = attacker.z - target.z;

        if (node == null) return CoverType.None;

        bool ge = false; // "greater or equal" rajalla?
        bool facesN = ge ? (dz >= 0) : (dz > 0);
        bool facesS = ge ? (dz <= 0) : (dz < 0);
        bool facesE = ge ? (dx >= 0) : (dx > 0);
        bool facesW = ge ? (dx <= 0) : (dx < 0);

        bool high =
            (facesN && node.HasHighCover(CoverMask.N)) ||
            (facesS && node.HasHighCover(CoverMask.S)) ||
            (facesE && node.HasHighCover(CoverMask.E)) ||
            (facesW && node.HasHighCover(CoverMask.W));

        if (high) return CoverType.High;

        bool low =
            (facesN && node.HasLowCover(CoverMask.N)) ||
            (facesS && node.HasLowCover(CoverMask.S)) ||
            (facesE && node.HasLowCover(CoverMask.E)) ||
            (facesW && node.HasLowCover(CoverMask.W));

        return low ? CoverType.Low : CoverType.None;
    }

    public static CoverType GetNodeAnyCover(PathNode node)
    {
    if (node == null) return CoverType.None;
```

```
    if (node.GetHighCoverMask() != CoverMask.None) return CoverType.High;
    if (node.GetLowCoverMask()  != CoverMask.None)  return CoverType.Low;
    return CoverType.None;
    }

}
```

Assets/scripts/CoverSystem/CoverVisualizer.cs

```
using UnityEngine;

[DefaultExecutionOrder(200)]
[DisallowMultipleComponent]
public class CoverVisualizer : MonoBehaviour
{
[Header("Refs")]
    [SerializeField] private PathFinding pathfinding;
    [SerializeField] private LevelGrid levelGrid;
    [SerializeField] private Camera cam;
    [SerializeField] private Material unlitTransparentMat; // Unlit/Transparent tms.

    [Header("Raycast")]
    [SerializeField] private LayerMask groundMask = ~0; // millä layereilla lattia/maa on

    [Header("Style")]
    [SerializeField] private float yOffset = 0.05f;    // nosta vähän lattiasta
    [SerializeField] private float edgeInset = 0.48f;  // 0.45–0.49
    [SerializeField] private float barLen = 0.90f;     // suhteessa cellSizeen
    [SerializeField] private float barWidth = 0.06f;   // X/Z -ohuus
    [SerializeField] private float barHeight = 0.06f;  // Y-paksuus
    [SerializeField] private Color lowColor  = new(0.2f, 1f, 0.2f, 0.55f);
    [SerializeField] private Color highColor = new(0.2f, 0.5f, 1f, 0.80f);

    [Header("Walls (optional)")]
    [SerializeField] private bool showWalls = true;
    [SerializeField] private Color wallColor = new(1f, 0.4f, 0.1f, 0.80f);

    Transform n,e,s,w; MeshRenderer rn,re,rs,rw; float cell;

    void Awake() {
        if (!pathfinding) pathfinding = FindFirstObjectByType<PathFinding>();
        if (!levelGrid)  levelGrid  = LevelGrid.Instance;
        if (!cam) cam = Camera.main;

        if (levelGrid == null) { enabled = false; return; }

        cell = levelGrid.GetCellSize();
        (n,rn) = CreateBar("N");
        (e,re) = CreateBar("E");
        (s,rs) = CreateBar("S");
        (w,rw) = CreateBar("W");
        HideAll();
    }

    (Transform, MeshRenderer) CreateBar(string name) {
        var go = GameObject.CreatePrimitive(PrimitiveType.Cube);
        go.name = $"CoverHover_{name}";
        Destroy(go.GetComponent<Collider>());
        go.transform.SetParent(transform, false);
```

```
        var mr = go.GetComponent<MeshRenderer>();
        if (unlitTransparentMat) mr.sharedMaterial = unlitTransparentMat;
        go.SetActive(false);
        return (go.transform, mr);
    }

    void Update() {
        BaseAction action = UnitActionSystem.Instance.GetSelectedAction();
        if (action == null) return;
        if (!pathfinding || !levelGrid || !cam || action.GetActionName() != "Move") { HideAll(); return; }

        var ray = cam.ScreenPointToRay(Input.mousePosition);
        if (!Physics.Raycast(ray, out var hit, 500f, groundMask, QueryTriggerInteraction.Collide)) { HideAll(); return; }

        // Ruudukkoon
        var gp = levelGrid.GetGridPosition(hit.point);
        var node = pathfinding.GetNode(gp.x, gp.z, gp.floor);
        if (node == null|| !node.GetIsWalkable()) { HideAll(); return; }

        var c = levelGrid.GetWorldPosition(gp);
        c.y += yOffset;

        // Reunan keskikohdat
        var north = c + new Vector3(0, 0,  cell * edgeInset);
        var south = c + new Vector3(0, 0, -cell * edgeInset);
        var eastP = c + new Vector3( cell * edgeInset, 0, 0);
        var westP = c + new Vector3(-cell * edgeInset, 0, 0);

        // N/S = pituus X-suunnassa, E/W = pituus Z-suunnassa
        DrawBar(node.HasHighCover(CoverMask.N), node.HasLowCover(CoverMask.N), node.HasWall(EdgeMask.N), n, rn, north, new Vector3(cell*barLen, barHeight, barWidth));
        DrawBar(node.HasHighCover(CoverMask.S), node.HasLowCover(CoverMask.S), node.HasWall(EdgeMask.S), s, rs, south, new Vector3(cell*barLen, barHeight, barWidth));
        DrawBar(node.HasHighCover(CoverMask.E), node.HasLowCover(CoverMask.E), node.HasWall(EdgeMask.E), e, re, eastP, new Vector3(barWidth, barHeight, cell*barLen));
        DrawBar(node.HasHighCover(CoverMask.W), node.HasLowCover(CoverMask.W), node.HasWall(EdgeMask.W), w, rw, westP, new Vector3(barWidth, barHeight, cell*barLen));
    }

    void DrawBar(bool high, bool low, bool wall, Transform tr, MeshRenderer mr, Vector3 pos, Vector3 size) {
        if (!high && !low && !(showWalls && wall)) { tr.gameObject.SetActive(false); return; }
        tr.gameObject.SetActive(true);
        tr.position = pos;
        tr.localScale = size;

        // Väri prioriteetilla: seinä > high cover > low cover
        var color = (showWalls && wall) ? wallColor : (high ? highColor : lowColor);
        var m = mr.material; // runtime-instanssi
        m.color = color;
    }

    void HideAll() {
        if (n) n.gameObject.SetActive(false);
        if (e) e.gameObject.SetActive(false);
        if (s) s.gameObject.SetActive(false);
        if (w) w.gameObject.SetActive(false);
```

```
    }
}
```

Assets/scripts/CoverSystem/EdgeBaker.cs

```
using System;
using System.Collections;
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; }

    // ==== TallWall (kapeat korkeat seinät) - asetukset ====
    [SerializeField] private LayerMask obstaclesMask;      // aseta Inspectorissa "Obstacles"
    [SerializeField] private float tallWallThresholdY = 1.8f;  // yli tämän = blokkaa LoS
    [SerializeField] private float edgeProbeHeight = 6f;       // kaistan korkeus
    /// ====


    [Header("References")]
    [SerializeField] private PathFinding pathfinding;
    [SerializeField] private LevelGrid levelGrid;

    [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;

    [Header("Cover scan")]
    [SerializeField] private LayerMask coverMask;

    [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;

    [Header("Cover height")]
    [SerializeField] private float lowCoverY = 1.0f;      // ~vyötärö
    [SerializeField] private float highCoverY = 1.6f;     // ~pää/olkapää

    // ---- 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;
    }

    private IEnumerator Start()
    {
        yield return new WaitUntil(() => LevelGrid.Instance != null && PathFinding.Instance != null);

        if (pathfinding == null) pathfinding = FindFirstObjectByType<PathFinding>();
        if (levelGrid == null) levelGrid = LevelGrid.Instance;

        Width = levelGrid.GetWidth();
        Height = levelGrid.GetHeight();
        FloorAmount = levelGrid.GetFloorAmount();
        CellSize = levelGrid.GetCellSize();

        if (GameModeManager.SelectedMode == GameMode.SinglePlayer && autoBakeOnStart)
        {
            BakeAllEdges();   // offline / yksinpeli
        }
    }

    // ------------------------- 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);
                }

        // 3) Update TallWall registry for LoS checks
        BakeTallWalls();
    }

    /// <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) || !IsWalkable(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 gridPosition)
    {
        var node = PF.GetNode(gridPosition.x, gridPosition.z, gridPosition.floor);
        node.ClearCover();


        // World-space center of this cell (at floor level)
        Vector3 center = LG.GetWorldPosition(gridPosition);
        float sellSize = CellSize;

        // Place the four strip centers exactly on the cell borders and lift to mid-scan height.
        float scanHeight = edgeScanHeight * 0.5f;
        Vector3 north = center + new Vector3(0f, scanHeight, +sellSize * 0.5f);
        Vector3 south = center + new Vector3(0f, scanHeight, -sellSize * 0.5f);
        Vector3 east = center + new Vector3(+sellSize * 0.5f, scanHeight, 0f);
        Vector3 west = center + new Vector3(-sellSize * 0.5f, scanHeight, 0f);

        PathBlocker(north, south, east, west, sellSize, node, gridPosition);
        WallCovers(north, south, east, west, sellSize, node, gridPosition);
    }

    private void PathBlocker(Vector3 north, Vector3 south, Vector3 east, Vector3 west, float sellSize, PathNode node, GridPosition gridPosition)
    {
        // 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(sellSize * edgeStripThickness * 0.5f, edgeScanHeight * 0.5f, sellSize * 0.45f);
        Vector3 halfEastWest = new(sellSize * 0.45f, edgeScanHeight * 0.5f, sellSize * edgeStripThickness * 0.5f);

        // 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(gridPosition, +0, +1, EdgeMask.S);
        }
        // Probe SOUTH edge; mirror to the southern neighbor.
        if (HasEdgeBlock(south, halfNorthSouth, Quaternion.identity))
        {
            node.AddWall(EdgeMask.S);
            MarkOpposite(gridPosition, +0, -1, EdgeMask.N);
        }
        // Probe EAST edge; mirror to the eastern neighbor.
        if (HasEdgeBlock(east, halfEastWest, Quaternion.identity))
        {
            node.AddWall(EdgeMask.E);
            MarkOpposite(gridPosition, +1, +0, EdgeMask.W);
        }
        // Probe WEST edge; mirror to the western neighbor.
        if (HasEdgeBlock(west, halfEastWest, Quaternion.identity))
        {
            node.AddWall(EdgeMask.W);
            MarkOpposite(gridPosition, -1, +0, EdgeMask.E);
        }
    }

    private void WallCovers(Vector3 north, Vector3 south, Vector3 east, Vector3 west, float sellSize, PathNode node, GridPosition gridPosition)
    {
        // --- Cover (sama geometria saa olla eri layerillä kuin edgeBlocker) ---
        // Tehdään matala ja korkea testi erikseen: low = vain vyötäröosuma, high = osuu myös pään korkeuteen.
        // Rajataan boksi vain yhdelle Y-korkeudelle (pieni korkeus), ettei pöydän jalat tms. vaikuta.
        Vector3 lowHalfNS = new Vector3(sellSize * edgeStripThickness * 0.5f, 0.1f, sellSize * 0.45f);
        Vector3 lowHalfEW = new Vector3(sellSize * 0.45f, 0.1f, sellSize * edgeStripThickness * 0.5f);
        Vector3 highHalfNS = lowHalfNS;
        Vector3 highHalfEW = lowHalfEW;

        // pisteet cover-korkeuksille
        Vector3 nLow = new Vector3(north.x, lowCoverY, north.z);
        Vector3 nHigh = new Vector3(north.x, highCoverY, north.z);
        Vector3 sLow = new Vector3(south.x, lowCoverY, south.z);
        Vector3 sHigh = new Vector3(south.x, highCoverY, south.z);
        Vector3 eLow = new Vector3(east.x, lowCoverY, east.z);
        Vector3 eHigh = new Vector3(east.x, highCoverY, east.z);
        Vector3 wLow = new Vector3(west.x, lowCoverY, west.z);
        Vector3 wHigh = new Vector3(west.x, highCoverY, west.z);

        // North
        bool nLowHit = Physics.CheckBox(nLow, lowHalfNS, Quaternion.identity, coverMask);
        bool nHighHit = Physics.CheckBox(nHigh, highHalfNS, Quaternion.identity, coverMask);
        if (nHighHit) node.AddHighCover(CoverMask.N);
        else if (nLowHit) node.AddLowCover(CoverMask.N);

        // South
        bool sLowHit = Physics.CheckBox(sLow, lowHalfNS, Quaternion.identity, coverMask);
        bool sHighHit = Physics.CheckBox(sHigh, highHalfNS, Quaternion.identity, coverMask);
        if (sHighHit) node.AddHighCover(CoverMask.S);
```

```
        else if (sLowHit) node.AddLowCover(CoverMask.S);

        // East
        bool eLowHit = Physics.CheckBox(eLow, lowHalfEW, Quaternion.identity, coverMask);
        bool eHighHit = Physics.CheckBox(eHigh, highHalfEW, Quaternion.identity, coverMask);
        if (eHighHit) node.AddHighCover(CoverMask.E);
        else if (eLowHit) node.AddLowCover(CoverMask.E);

        // West
        bool wLowHit = Physics.CheckBox(wLow, lowHalfEW, Quaternion.identity, coverMask);
        bool wHighHit = Physics.CheckBox(wHigh, highHalfEW, Quaternion.identity, coverMask);
        if (wHighHit) node.AddHighCover(CoverMask.W);
        else if (wLowHit) node.AddLowCover(CoverMask.W);
    }

    /// <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;
    }

    public void BakeTallWalls()
    {

        var lg = LevelGrid.Instance;
        if (lg == null) return;

        EdgeOcclusion.Clear();

        int w = lg.GetWidth();
        int h = lg.GetHeight();
        int floors = lg.GetFloorAmount();

        float cell = lg.GetCellSize();
        float halfCell = cell * 0.5f;

        float castDistance = 0.3f;
        float boxThickness = cell * 0.8f;
        float boxHeight = edgeProbeHeight;
        float boxDepth = 0.1f;

        var boxExtN = new UnityEngine.Vector3(boxThickness * 0.5f, boxHeight * 0.5f, boxDepth * 0.5f);
        var boxExtE = new UnityEngine.Vector3(boxDepth * 0.5f, boxHeight * 0.5f, boxThickness * 0.5f);

        for (int f = 0; f < floors; f++)
        {
            for (int z = 0; z < h; z++)
```

```
        {
            for (int x = 0; x < w; x++)
            {
                var gp = new GridPosition(x, z, f);
                var basePos = lg.GetWorldPosition(gp);
                float baseY = basePos.y;

                // N-reuna (z+ suuntaan)
                {
                    var startPos = basePos + new UnityEngine.Vector3(0f, boxHeight * 0.5f, halfCell - castDistance * 0.5f);
                    var direction = UnityEngine.Vector3.forward;

                    UnityEngine.RaycastHit[] hits = UnityEngine.Physics.BoxCastAll(
                        startPos, boxExtN, direction, UnityEngine.Quaternion.identity,
                        castDistance, obstaclesMask, UnityEngine.QueryTriggerInteraction.Ignore);

                    float maxTopAboveBase = 0f;
                    foreach (var hit in hits)
                    {
                        float topRel = hit.collider.bounds.max.y - baseY;
                        if (topRel > maxTopAboveBase) maxTopAboveBase = topRel;
                    }

                    if (maxTopAboveBase >= tallWallThresholdY)
                    {
                        EdgeOcclusion.AddSymmetric(gp, EdgeMask.N);
                    }
                }

                // E-reuna (x+ suuntaan)
                {
                    var startPos = basePos + new UnityEngine.Vector3(halfCell - castDistance * 0.5f, boxHeight * 0.5f, 0f);
                    var direction = UnityEngine.Vector3.right;

                    UnityEngine.RaycastHit[] hits = UnityEngine.Physics.BoxCastAll(
                        startPos, boxExtE, direction, UnityEngine.Quaternion.identity,
                        castDistance, obstaclesMask, UnityEngine.QueryTriggerInteraction.Ignore);

                    float maxTopAboveBase = 0f;
                    foreach (var hit in hits)
                    {
                        float topRel = hit.collider.bounds.max.y - baseY;
                        if (topRel > maxTopAboveBase) maxTopAboveBase = topRel;
                    }

                    if (maxTopAboveBase >= tallWallThresholdY)
                    {
                        EdgeOcclusion.AddSymmetric(gp, EdgeMask.E);
                    }
                }
            }
        }
```

```
        }

        int totalEdges = 0;
        for (int f = 0; f < floors; f++)
        {
            for (int z = 0; z < h; z++)
            {
                for (int x = 0; x < w; x++)
                {
                    var gp = new GridPosition(x, z, f);
                    if (EdgeOcclusion.HasTallWall(gp, EdgeMask.N)) totalEdges++;
                    if (EdgeOcclusion.HasTallWall(gp, EdgeMask.E)) totalEdges++;
                    if (EdgeOcclusion.HasTallWall(gp, EdgeMask.S)) totalEdges++;
                    if (EdgeOcclusion.HasTallWall(gp, EdgeMask.W)) totalEdges++;
                }
            }
        }
    }

}
```

Assets/scripts/CoverSystem/EdgeOcclusion.cs

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


/// <summary>
/// TallWall-occlusion per ruutu: mitkä reunat (N/E/S/W) ovat "korkeita kapeita seiniä".
/// Tallennetaan vain ruudut joissa on vähintään yksi bitti.
/// </summary>
public static class EdgeOcclusion
{
    private static readonly Dictionary<GridPosition, EdgeMask> _tallWalls = new();

    public static void Clear() => _tallWalls.Clear();

    public static bool HasTallWall(GridPosition cell, EdgeMask side)
    {
        if (!_tallWalls.TryGetValue(cell, out var m)) return false;
        return (m & side) != 0;
    }

    public static void AddSymmetric(GridPosition cell, EdgeMask side)
    {
        // aseta celliin
        if (_tallWalls.TryGetValue(cell, out var m)) _tallWalls[cell] = m | side;
        else _tallWalls[cell] = side;

        // aseta myös naapuriin vastakkaiselle reunalle
        var lg = LevelGrid.Instance;
        if (lg == null) return;

        GridPosition n; EdgeMask opposite;
        switch (side)
        {
            case EdgeMask.N: n = new GridPosition(cell.x, cell.z + 1, cell.floor); opposite = EdgeMask.S; break;
            case EdgeMask.E: n = new GridPosition(cell.x + 1, cell.z, cell.floor); opposite = EdgeMask.W; break;
            case EdgeMask.S: n = new GridPosition(cell.x, cell.z - 1, cell.floor); opposite = EdgeMask.N; break;
            case EdgeMask.W: n = new GridPosition(cell.x - 1, cell.z, cell.floor); opposite = EdgeMask.E; break;
            default: return;
        }
        if (!lg.IsValidGridPosition(n)) return;

        if (_tallWalls.TryGetValue(n, out var nm)) _tallWalls[n] = nm | opposite;
        else _tallWalls[n] = opposite;
    }
}
```

Assets/scripts/DebuggingAndTesting/GridDebug/CoverDebugGizmos.cs

```
using UnityEngine;

[ExecuteAlways]
[DisallowMultipleComponent]
public class CoverDebugGizmos : MonoBehaviour
{
    [Header("Refs")]
    [SerializeField] private PathFinding pathfinding;
    [SerializeField] private LevelGrid levelGrid;

    [Header("Filter")]
    [Tooltip("Piirretään vain tämä kerros (floor).")]
    [SerializeField] private int visibleFloor = 0;
    [SerializeField] private bool drawOnlyVisibleFloor = true;

    [Header("What to draw")]
    [SerializeField] private bool drawWalls = true;
    [SerializeField] private bool drawLowCover = true;
    [SerializeField] private bool drawHighCover = true;

    [Header("Style")]
    [SerializeField] private float yOffset = 0.05f;     // nosta viivaa vähän lattiasta
    [SerializeField] private float edgeInset = 0.48f;   // kuinka lähellä ruudun reunaa (0.5 = ihan reunalla)
    [SerializeField] private float wallThickness = 0.08f;
    [SerializeField] private float coverThickness = 0.05f;
    [SerializeField] private float coverLength = 0.35f; // viivan pituus reunan suuntaisesti

    [Header("Colors")]
    [SerializeField] private Color wallColor = new Color(1f, 0.4f, 0.1f, 0.9f); // oranssi
    [SerializeField] private Color lowColor = new Color(0.2f, 1f, 0.2f, 0.9f); // vihreä
    [SerializeField] private Color highColor = new Color(0.2f, 0.5f, 1f, 0.9f); // sininen

    private PathFinding PF => pathfinding ? pathfinding : (pathfinding = FindFirstObjectByType<PathFinding>());
    private LevelGrid LG => levelGrid ? levelGrid : (levelGrid = LevelGrid.Instance);

    private void OnDrawGizmos()
    {
        if (PF == null || LG == null) return;

        int width = PF.GetWidth();
        int height = PF.GetHeight();
        int floors = LG.GetFloorAmount();
        float s = LG.GetCellSize();

        for (int f = 0; f < floors; f++)
        {
            if (drawOnlyVisibleFloor && f != visibleFloor) continue;

            for (int x = 0; x < width; x++)
                for (int z = 0; z < height; z++)
```

```
            {
                var node = PF.GetNode(x, z, f);
                if (node == null) continue;

                Vector3 c = LG.GetWorldPosition(new GridPosition(x, z, f));
                c.y += yOffset;

                // TESTI: piirrä pieni pallo jos ruudulla on coveria
                if (node.GetHighCoverMask() != 0 || node.GetLowCoverMask() != 0)
                {
                    Gizmos.color = Color.cyan;
                    Gizmos.DrawSphere(c + Vector3.up * 0.2f, 0.05f);
                }

                // Reunakohdat (keskitettyinä reunoille)
                Vector3 n = c + new Vector3(0, 0, +s * edgeInset);
                Vector3 s_ = c + new Vector3(0, 0, -s * edgeInset);
                Vector3 e = c + new Vector3(+s * edgeInset, 0, 0);
                Vector3 w = c + new Vector3(-s * edgeInset, 0, 0);

                // Seinät
                if (drawWalls)
                {
                    Gizmos.color = wallColor;
                    if (node.HasWall(EdgeMask.N)) DrawEdgeBar(n, Vector3.right, wallThickness, s * 0.9f);
                    if (node.HasWall(EdgeMask.S)) DrawEdgeBar(s_, Vector3.right, wallThickness, s * 0.9f);
                    if (node.HasWall(EdgeMask.E)) DrawEdgeBar(e, Vector3.forward, wallThickness, s * 0.9f);
                    if (node.HasWall(EdgeMask.W)) DrawEdgeBar(w, Vector3.forward, wallThickness, s * 0.9f);
                }

                // Cover (valinnainen: toimii, jos lisäsit CoverMaskin PathNodeen)
                if (drawLowCover)
                {
                    Gizmos.color = lowColor;
                    if (node.HasLowCover(CoverMask.N)) DrawEdgeBar(n, Vector3.right, coverThickness, s * coverLength);
                    if (node.HasLowCover(CoverMask.S)) DrawEdgeBar(s_, Vector3.right, coverThickness, s * coverLength);
                    if (node.HasLowCover(CoverMask.E)) DrawEdgeBar(e, Vector3.forward, coverThickness, s * coverLength);
                    if (node.HasLowCover(CoverMask.W)) DrawEdgeBar(w, Vector3.forward, coverThickness, s * coverLength);
                }

                if (drawHighCover)
                {
                    Gizmos.color = highColor;
                    if (node.HasHighCover(CoverMask.N)) DrawEdgeBar(n + Vector3.up * 0.02f, Vector3.right, coverThickness, s * coverLength);
                    if (node.HasHighCover(CoverMask.S)) DrawEdgeBar(s_ + Vector3.up * 0.02f, Vector3.right, coverThickness, s * coverLength);
                    if (node.HasHighCover(CoverMask.E)) DrawEdgeBar(e + Vector3.up * 0.02f, Vector3.forward, coverThickness, s * coverLength);
                    if (node.HasHighCover(CoverMask.W)) DrawEdgeBar(w + Vector3.up * 0.02f, Vector3.forward, coverThickness, s * coverLength);
                }
            }
        }
    }
```

```
    // Piirtää “paksun viivan” reunan suuntaisesti pienenä laatikkona
    private void DrawEdgeBar(Vector3 center, Vector3 along, float thickness, float length)
    {
        // along = joko Vector3.right (itä-länsi) tai Vector3.forward (pohjois-etelä)
        Vector3 size = new Vector3(
            Mathf.Abs(along.x) > 0 ? length : thickness,
            thickness,
            Mathf.Abs(along.z) > 0 ? length : thickness
        );
        Gizmos.DrawCube(center, size);
    }
}
```

Assets/scripts/DebuggingAndTesting/GridDebug/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/DebuggingAndTesting/GridDebug/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/DebuggingAndTesting/PathfindingDebug/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/PathfindingDebug/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()
    {
        var m = (PathfindingLinkMonoBehaviour)target;
        var t = m.transform;

        // Local -> World kahvoille
        Vector3 aW = t.TransformPoint(m.linkPositionA);
        Vector3 bW = t.TransformPoint(m.linkPositionB);

        EditorGUI.BeginChangeCheck();
        Vector3 naW = Handles.PositionHandle(aW, Quaternion.identity);
        Vector3 nbW = Handles.PositionHandle(bW, Quaternion.identity);
        if (EditorGUI.EndChangeCheck())
        {
            Undo.RecordObject(m, "Change Link Position");
            // World -> Local talteen
            m.linkPositionA = t.InverseTransformPoint(naW);
            m.linkPositionB = t.InverseTransformPoint(nbW);
        }
    }
}
```

Assets/scripts/Editor/UnusedMonoBehavioursReporter.cs

```
#if UNITY_EDITOR
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using UnityEditor;
using UnityEditor.SceneManagement;
using UnityEngine;

namespace RogueShooter.Tools
{
    /// <summary>
    /// Reports MonoBehaviour scripts that do not appear in any prefab or scene dependencies.
    /// Limitations:
    /// - Does NOT detect scripts that are only added at runtime via AddComponent/Reflection.
    /// - Excludes scripts under any *Editor* folder by default.
    /// - Skips abstract classes and classes that don't inherit MonoBehaviour.
    /// - Prefabs & scenes outside the current project (Packages, external) are not scanned.
    /// </summary>
    public class UnusedMonoBehavioursReporter : EditorWindow
    {

        private const string ScanRoot = "Assets/Scripts";

        [MenuItem("Tools/RogueShooter/Report Unused MonoBehaviours")]
        public static void Open()
        {
            var win = GetWindow<UnusedMonoBehavioursReporter>(true, "Unused MonoBehaviours Report", true);
            win.minSize = new Vector2(820, 480);
            win.RefreshScan();
        }

        private Vector2 _scroll;
        private string _search = string.Empty;
        private bool _includeEditorFolderScripts = false; // Usually false; keep Editor scripts out
        private bool _showUsedAlso = false;

        private List<ScriptInfo> _allCandidates = new();   // All MonoBehaviour scripts in project (filtered)
        private HashSet<string> _usedScriptPaths = new(StringComparer.OrdinalIgnoreCase); // .cs paths used by scenes/prefabs
        private List<ScriptInfo> _unused = new();           // Candidates not referenced by any scene/prefab
        private List<ScriptInfo> _used = new();             // Candidates referenced
        private string _status = "";
        private double _elapsedMs;

        private class ScriptInfo
        {
            public MonoScript Mono;
            public Type Type;
            public string Path;
```

```
            public string AssemblyName;
            public bool IsAbstract;
            public bool InEditorFolder;
        }

        private void OnGUI()
        {
            GUILayout.Space(6);

            using (new EditorGUILayout.HorizontalScope())
            {
                _includeEditorFolderScripts = GUILayout.Toggle(_includeEditorFolderScripts, new GUIContent("Include /Editor scripts"), GUILayout.Width(170));
                _showUsedAlso = GUILayout.Toggle(_showUsedAlso, new GUIContent("Show used scripts too"), GUILayout.Width(170));
                GUILayout.Space(12);

                GUILayout.Label("Search:", GUILayout.Width(48));
                _search = GUILayout.TextField(_search ?? string.Empty);
                GUILayout.FlexibleSpace();

                if (GUILayout.Button("Refresh", GUILayout.Width(100)))
                    RefreshScan();

                if (GUILayout.Button("Export CSV...", GUILayout.Width(120)))
                    ExportCsv();
            }

            GUILayout.Space(4);
            EditorGUILayout.LabelField($"Scan root: {ScanRoot}");
            EditorGUILayout.HelpBox("Lists MonoBehaviour scripts that are NOT referenced by any prefab or scene in this project. " +
                                    "Dynamic AddComponent() usage will not be detected.", MessageType.Info);

            if (!string.IsNullOrEmpty(_status))
                EditorGUILayout.LabelField(_status);
            EditorGUILayout.LabelField($"Scanned in {_elapsedMs:F0} ms | Candidates: {_allCandidates.Count} | Used: {_used.Count} | Unused: {_unused.Count}");

            GUILayout.Space(6);

            using (var scroll = new EditorGUILayout.ScrollViewScope(_scroll))
            {
                _scroll = scroll.scrollPosition;

                DrawTableHeader();
                DrawList(_unused, title: "UNUSED SCRIPTS", color: new Color(1f, 0.5f, 0.5f));
                if (_showUsedAlso)
                {
                    GUILayout.Space(6);
                    DrawList(_used, title: "USED SCRIPTS", color: new Color(0.65f, 0.9f, 0.65f));
                }
            }
        }

        private void DrawTableHeader()
```

```
        {
            var rect = GUILayoutUtility.GetRect(10, 24, GUILayout.ExpandWidth(true));
            EditorGUI.DrawRect(rect, new Color(0.18f, 0.18f, 0.18f));
            var r = rect; r.x += 6; r.width -= 12; r.y += 3;
            GUI.Label(r, "Name | Namespace | Assembly | Path");
        }

        private void DrawList(List<ScriptInfo> list, string title, Color color)
        {
            var shown = ApplySearch(list);

            var head = GUILayoutUtility.GetRect(10, 24, GUILayout.ExpandWidth(true));
            EditorGUI.DrawRect(head, color);
            var hr = head; hr.x += 6; hr.width -= 12; hr.y += 3;
            GUI.Label(hr, $"{title} ({shown.Count})");

            foreach (var s in shown)
            {
                var row = GUILayoutUtility.GetRect(10, 22, GUILayout.ExpandWidth(true));
                if (Event.current.type == EventType.MouseDown && row.Contains(Event.current.mousePosition))
                {
                    Selection.activeObject = s.Mono; // select asset
                    EditorGUIUtility.PingObject(s.Mono);
                    Repaint();
                }

                EditorGUI.DrawRect(row, new Color(0.12f, 0.12f, 0.12f));
                var rr = row; rr.x += 6; rr.width -= 12; rr.y += 2;
                var name = s.Type != null ? s.Type.Name : Path.GetFileNameWithoutExtension(s.Path);
                var ns = s.Type != null ? (string.IsNullOrEmpty(s.Type.Namespace) ? "-" : s.Type.Namespace) : "-";
                var asm = s.AssemblyName ?? "-";
                GUI.Label(rr, $"{name} | {ns} | {asm} | {s.Path}");
            }
        }

        private List<ScriptInfo> ApplySearch(List<ScriptInfo> src)
        {
            if (string.IsNullOrWhiteSpace(_search)) return src;
            var q = _search.Trim();
            return src.Where(s =>
                    (s.Type != null && (s.Type.Name.IndexOf(q, StringComparison.OrdinalIgnoreCase) >= 0 ||
                                        (!string.IsNullOrEmpty(s.Type.Namespace) && s.Type.Namespace.IndexOf(q, StringComparison.OrdinalIgnoreCase) >= 0)))
                    || (!string.IsNullOrEmpty(s.Path) && s.Path.IndexOf(q, StringComparison.OrdinalIgnoreCase) >= 0)
                ).ToList();
        }

        private static bool IsUnderRoot(string assetPath)
        {
            return !string.IsNullOrEmpty(assetPath) && assetPath.StartsWith(ScanRoot, StringComparison.OrdinalIgnoreCase);
        }

        private void RefreshScan()
```

```
    {
        try
        {
            EditorUtility.DisplayProgressBar("Scanning", "Collecting scripts...", 0f);
            var sw = Stopwatch.StartNew();

            _allCandidates = CollectCandidateScripts(_includeEditorFolderScripts);

            EditorUtility.DisplayProgressBar("Scanning", "Collecting prefab/scene dependencies...", 0.33f);
            _usedScriptPaths = CollectUsedScriptPaths();

            EditorUtility.DisplayProgressBar("Scanning", "Comparing...", 0.66f);
            _unused = new List<ScriptInfo>();
            _used = new List<ScriptInfo>();

            foreach (var s in _allCandidates)
            {
                if (_usedScriptPaths.Contains(s.Path)) _used.Add(s);
                else _unused.Add(s);
            }

            sw.Stop();
            _elapsedMs = sw.Elapsed.TotalMilliseconds;
            _status = $"Found {_unused.Count} potentially unused MonoBehaviour scripts.";
        }
        catch (Exception e)
        {
            UnityEngine.Debug.LogError($"[UnusedMonoBehavioursReporter] Scan failed: {e}\n{e.StackTrace}");
            _status = "Scan failed – see Console.";
        }
        finally
        {
            EditorUtility.ClearProgressBar();
            Repaint();
        }
    }

    private static List<ScriptInfo> CollectCandidateScripts(bool includeEditorFolder)
    {
        var results = new List<ScriptInfo>();
        var guids = AssetDatabase.FindAssets("t:MonoScript");

        for (int i = 0; i < guids.Length; i++)
        {
            if (i % 200 == 0)
                EditorUtility.DisplayProgressBar("Scanning", $"Inspecting scripts ({i}/{guids.Length})...", Mathf.InverseLerp(0, guids.Length, i));

            var path = AssetDatabase.GUIDToAssetPath(guids[i]);
            if (string.IsNullOrEmpty(path) || !IsUnderRoot(path)) continue;
            var mono = AssetDatabase.LoadAssetAtPath<MonoScript>(path);
            if (mono == null) continue;
```

```
            var type = mono.GetClass();
            if (type == null) continue; // no class in file or compile error

            if (!typeof(MonoBehaviour).IsAssignableFrom(type))
                continue; // not a MonoBehaviour

            if (type.IsAbstract)
                continue; // we report only concrete components

            var inEditorFolder = path.Split('/')
                                     .Any(seg => string.Equals(seg, "Editor", StringComparison.OrdinalIgnoreCase));
            if (inEditorFolder && !includeEditorFolder)
                continue; // ignore Editor scripts unless explicitly included

            var asmName = type.Assembly?.GetName().Name;

            results.Add(new ScriptInfo
            {
                Mono = mono,
                Type = type,
                Path = path,
                AssemblyName = asmName,
                IsAbstract = false,
                InEditorFolder = inEditorFolder,
            });
        }

        return results;
    }

    private static HashSet<string> CollectUsedScriptPaths()
    {
        var used = new HashSet<string>(StringComparer.OrdinalIgnoreCase);

        // Gather all prefab & scene asset paths
        var prefabGuids = AssetDatabase.FindAssets("t:Prefab");
        var sceneGuids  = AssetDatabase.FindAssets("t:Scene");

        // Unity's dependency graph: scenes/prefabs depend on MonoScript assets referenced by m_Script
        void Accumulate(string guid, int index, int total, string label)
        {
            if (index % 100 == 0)
                EditorUtility.DisplayProgressBar("Scanning", $"{label} ({index}/{total})...", Mathf.InverseLerp(0, total, index));

            var path = AssetDatabase.GUIDToAssetPath(guid);
            if (string.IsNullOrEmpty(path)) return;

            var deps = AssetDatabase.GetDependencies(path, true);
            foreach (var d in deps)
            {
                if (d.EndsWith(".cs", StringComparison.OrdinalIgnoreCase))
                    used.Add(d);
```

```
                }
            }

            for (int i = 0; i < prefabGuids.Length; i++)
                Accumulate(prefabGuids[i], i, prefabGuids.Length, "Scanning prefabs");

            for (int i = 0; i < sceneGuids.Length; i++)
                Accumulate(sceneGuids[i], i, sceneGuids.Length, "Scanning scenes");

            return used;
        }

        private void ExportCsv()
        {
            try
            {
                var path = EditorUtility.SaveFilePanelInProject("Export CSV", "UnusedMonoBehaviours.csv", "csv", "Select save location");
                if (string.IsNullOrEmpty(path)) return;

                var lines = new List<string> { "Name,Namespace,Assembly,AssetPath" };
                foreach (var s in ApplySearch(_unused))
                {
                    var name = s.Type != null ? s.Type.Name : Path.GetFileNameWithoutExtension(s.Path);
                    var ns = s.Type != null ? (s.Type.Namespace ?? string.Empty) : string.Empty;
                    var asm = s.AssemblyName ?? string.Empty;

                    string Escape(string x) => '"' + (x?.Replace("\"", "\"\"") ?? string.Empty) + '"';
                    lines.Add(string.Join(",", new[] { Escape(name), Escape(ns), Escape(asm), Escape(s.Path) }));
                }

                File.WriteAllLines(path, lines);
                AssetDatabase.ImportAsset(path);
                EditorUtility.RevealInFinder(path);
            }
            catch (Exception e)
            {
                UnityEngine.Debug.LogError($"[UnusedMonoBehavioursReporter] Export failed: {e}");
                EditorUtility.DisplayDialog("Export CSV", "Export failed. See Console.", "OK");
            }
        }
    }
}
#endif
```

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 Unit actingUnit;
    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;
        }

        Unit.OnAnyUnitDead += HandleAnyUnitDead;

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

        Unit.OnAnyUnitDead -= HandleAnyUnitDead;
    }

    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())
        {
            //NOTE! Just for testing. AI not do this for now.
            if (baseAction.GetActionName() == "Overwatch")
            {
                Debug.Log("[Enemy AI] I am too dumd to do Overwatch action!");
                // Enemy AI Cant handle this action right now.
                continue;
            }

            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))
        {
            actingUnit = enemyUnit;
            // bestBaseAction.TakeAction(bestEnemyAIAction.gridPosition, onEnemyAIActionComplete);
            bestBaseAction.TakeAction(bestEnemyAIAction.gridPosition, OnEnemyAIActionComplete);
            return true;
        }
        else
```

```
        {
            return false;
        }
    }

    private void OnEnemyAIActionComplete()
    {
        actingUnit = null;
        SetStateTakingTurn();              // palaa Busy -> TakingTurn
    }

    private void HandleAnyUnitDead(object sender, EventArgs e)
    {
        var dead = sender as Unit;                 // sender on se Unit joka kuoli
        if (dead == null) return;

        if (state != State.Busy) return;           // reagoidaan vain jos AI odottaa actionia
        if (actingUnit == null) return;               // ei käynnissä olevaa AI-yksikköä
        if (dead != actingUnit) return;               // kuollut ei ollut juuri se joka toimi

        actingUnit= null;
        UnitActionSystem.Instance.SetBusy();
        SetStateTakingTurn();                      // jatka AI-kierrosta
    }

    /// <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/GameBalance/GameBalance.cs

```
using UnityEngine;

[DefaultExecutionOrder(-1000)]
public class GameBalance : MonoBehaviour {
    public static GameBalance I { get; private set; }
    [SerializeField] CombatRanges ranges;
    public static CombatRanges R => I ? I.ranges : null;

    void Awake() {
        if (I && I != this) { Destroy(gameObject); return; }
        I = this;
        DontDestroyOnLoad(gameObject);
        if (ranges == null)
            Debug.LogWarning("GameBalance.ranges puuttuu — käytetään Resources-fallbackia jos saatavilla.");
    }
}
```

Assets/scripts/GameLogic/BattleLogic/TurnSystem.cs

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

public class TurnSystem : MonoBehaviour
{
    public static TurnSystem Instance { get; private set; }
    public Team CurrentTeam { get; set; } = Team.Player;
    public int TurnId { get; set; } = 0;

    public event Action<Team,int> OnTurnStarted;
    public event Action<Team,int> OnTurnEnded;

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

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

    private void Start()
    {
        OnTurnStarted += turnSystem_OnTurnStarted;
        OnTurnEnded += turnSystem_OnTurnEnded;

        OnTurnChanged += turnSystem_OnTurnChanged;

        StartCoroutine(Co_DeferredFirstTurnKick());
    }

    private void OnDisable()
    {
        OnTurnChanged -= turnSystem_OnTurnChanged;
        OnTurnStarted -= turnSystem_OnTurnStarted;
        OnTurnEnded -= turnSystem_OnTurnEnded;
    }

    private void turnSystem_OnTurnChanged(object sender, EventArgs e)
    {
        UnitActionSystem.Instance.ResetSelectedAction();
        UnitActionSystem.Instance.ResetSelectedUnit();
```

```
    }

    private System.Collections.IEnumerator Co_DeferredFirstTurnKick()
    {
        // odota 1–2 framea että UnitManager, UnitActionSystem, StatusCoordinator ym. ovat varmasti ylhäällä
        yield return null;
        yield return null;

        OnTurnStarted?.Invoke(CurrentTeam, TurnId);
        PlayerLocalTurnGate.Set(isPlayerTurn);
        OnTurnChanged?.Invoke(this, EventArgs.Empty);
    }

    private void turnSystem_OnTurnStarted(Team startTurnTeam, int turnId)
    {
        if (NetMode.IsRemoteClient)
    {
        // Client: päivitä vain oman tiimin unitit
        StartCoroutine(Co_ClientTurnStart(startTurnTeam));
        return;
    }

    // Host/Server/Offline: normaali logiikka
    StartCoroutine(Co_SafeTurnStart(startTurnTeam, turnId));
    }

    private System.Collections.IEnumerator Co_SafeTurnStart(Team startTurnTeam, int turnId)
    {
        const int MAX_WAIT_FRAMES = 180;
        int waitedFrames = 0;

        // 1) Odota, että mikään action ei ole aktiivinen
        while (BaseAction.AnyActionActive() && waitedFrames < MAX_WAIT_FRAMES)
        {
            waitedFrames++;
            yield return null;
        }

        if (BaseAction.AnyActionActive())
        {
            Debug.LogWarning("[TurnSystem] Actionit ei päättyneet ajoissa, pakkolopetetaan!");
            BaseAction.ForceCompleteAllActiveActions();
            yield return null;
        }

        // 2) Odota kriittiset singletonit
        while ((UnitManager.Instance == null || UnitActionSystem.Instance == null || StatusCoordinator.Instance == null)
            && waitedFrames < MAX_WAIT_FRAMES)
        {
            waitedFrames++;
            yield return null;
        }
```

```
        // 3) Jos ollaan verkossa serverinä/hostina ja käytetään vision-RPC:tä, odota agentti
        if (Mirror.NetworkServer.active && !NetworkSync.IsOffline)
        {
            int w = 0;
            while (NetworkSyncAgent.Local == null && w < 60) { w++; yield return null; }
        }

        // 4) Avaa input
        UnitActionSystem.Instance.UnlockInput();

        // 5) Kerää vuoron aloittavat unitit
        List<Unit> units = new();
        var all = UnitManager.Instance?.GetAllUnitList();
        if (all != null)
        {
            foreach (Unit unit in all)
            {
                if (!unit) continue;
                if (unit.Team != startTurnTeam) continue;

                units.Add(unit);

                // Voi puuttua rootista – tee null-guard:
                var ba = unit.GetComponent<BaseAction>();
                if (ba != null) ba.ResetChostActions();

                //Resetoi Overwach shootaction bool asetus.
                unit.GetComponent<ShootAction>().ResetOverwatchShotState();
            }
        }

        // 6) Rebuild team vision (start phase):
        //    - offline: tee lokaalisti
        //    - host/server: RPC koko tiimille

        int teamID = (startTurnTeam == Team.Player) ? 0 : 1;  // yksiselitteinen mappi tähän

        // 7) Tyhjennä overwatch-visuaalit kaikilla clienteilla
        if (!NetworkSync.IsOffline && Mirror.NetworkServer.active && NetworkSyncAgent.Local != null)
        {
            NetworkSyncAgent.Local.RpcClearAllOverwatchVisuals(teamID);
        }

        // 8) Ilmoita statusit
        StatusCoordinator.Instance.UnitTurnStartStatus(units);

        yield return null;

        if (NetworkSync.IsOffline)
            TeamVisionService.Instance.RebuildTeamVisionLocal(teamID, false);
        else if (Mirror.NetworkServer.active && NetworkSyncAgent.Local != null)
```

```
            NetworkSyncAgent.Local.ServerPushTeamVision(teamID, endPhase: false);


    }

    private System.Collections.IEnumerator Co_ClientTurnStart(Team startTurnTeam)
    {
        // Odota että actionit päättyy
        const int MAX_WAIT_FRAMES = 180;
        int waitedFrames = 0;

        while (BaseAction.AnyActionActive() && waitedFrames < MAX_WAIT_FRAMES)
        {
            waitedFrames++;
            yield return null;
        }

        if (BaseAction.AnyActionActive())
        {
            Debug.LogWarning("[TurnSystem-Client] Actionit ei päättyneet, pakkolopetetaan!");
            BaseAction.ForceCompleteAllActiveActions();
            yield return null;
        }

        // Odota singletonit
        while ((UnitManager.Instance == null || StatusCoordinator.Instance == null) && waitedFrames < MAX_WAIT_FRAMES)
        {
            waitedFrames++;
            yield return null;
        }

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

        // Kerää oman tiimin unitit ja päivitä niiden näkyvyys
        List<Unit> units = new();
        var all = UnitManager.Instance?.GetAllUnitList();
        if (all != null)
        {
            foreach (Unit unit in all)
            {
                if (!unit) continue;
                if (unit.Team != startTurnTeam) continue;

                units.Add(unit);

                var ba = unit.GetComponent<BaseAction>();
                if (ba != null) ba.ResetChostActions();
```

```
                unit.GetComponent<ShootAction>().ResetOverwatchShotState();
            }
        }

        // KRIITTINEN: Päivitä unitien näkyvyys 360°
        if (StatusCoordinator.Instance != null)
        {
            StatusCoordinator.Instance.UnitTurnStartStatus(units);
        }

        yield return null; // anna yhden framen hengähdys

        int myTeam = (startTurnTeam == Team.Player) ? 0 : 1;
        if (TeamVisionService.Instance != null)
            TeamVisionService.Instance.RebuildTeamVisionLocal(myTeam, endPhase:false);
    }


    public void NextTurn()
    {

        if (GameModeManager.SelectedMode != GameMode.SinglePlayer && !NetMode.IsOnline)
        {
            Debug.LogWarning("Client yritti kääntää vuoroa lokaalisti, ignoroidaan.");
            return;
        }

        OnTurnEnded?.Invoke(CurrentTeam, TurnId);
        CurrentTeam = (CurrentTeam == Team.Player) ? Team.Enemy : Team.Player;
        TurnId++;
        OnTurnStarted?.Invoke(CurrentTeam, TurnId);

        if (GameModeManager.SelectedMode == GameMode.SinglePlayer)
        {
            turnNumber++;
            isPlayerTurn = !isPlayerTurn;
            OnTurnChanged?.Invoke(this, EventArgs.Empty);
            PlayerLocalTurnGate.Set(isPlayerTurn);
        }
        else if (GameModeManager.SelectedMode == GameMode.CoOp)
        {

        }
        else if (GameModeManager.SelectedMode == GameMode.Versus)
        {

        }
    }

    private void turnSystem_OnTurnEnded(Team endTurnTeam, int turnId)
    {
        if (NetMode.IsRemoteClient) return;
```

```
        // rebuild end-phase vision: kartio
        int teamId = (endTurnTeam == Team.Player) ? 0 : 1;

        if (NetworkSync.IsOffline)
            TeamVisionService.Instance.RebuildTeamVisionLocal(teamId, true);
        // RebuildTeamVisionLocal(teamId, endPhase: true);
        else if (Mirror.NetworkServer.active && NetworkSyncAgent.Local != null)
            NetworkSyncAgent.Local.ServerPushTeamVision(teamId, endPhase: true);

        // lista vuoron lopettavista
        List<Unit> units = new();
        var all = UnitManager.Instance?.GetAllUnitList();
        if (all != null)
        {
            foreach (Unit unit in all)
            {
                if (!unit) continue;
                if (unit.Team != endTurnTeam) continue;
                units.Add(unit);
            }
        }

        Debug.Log($"[TurnSystem] Calling UnitTurnEndStatus, ServerActive={Mirror.NetworkServer.active}");
        StatusCoordinator.Instance.UnitTurnEndStatus(units);
    }

    public void ForcePhase(bool isPlayerTurn, bool incrementTurnNumber)
    {
        if (incrementTurnNumber) turnNumber++;

        /*
        if (NetMode.IsOnline && isPlayerTurn)
        {
            ConvertUnusedActionPointsToCoverPoints();
        }
        */
        this.isPlayerTurn = isPlayerTurn;
        OnTurnChanged?.Invoke(this, EventArgs.Empty);
    }

    public void SetHudFromNetwork(int newTurnNumber, bool isPlayersPhase)
    {
        turnNumber = newTurnNumber;
        isPlayerTurn = isPlayersPhase;
        OnTurnChanged?.Invoke(this, EventArgs.Empty);
    }

    private void ConvertUnusedActionPointsToCoverPoints()
    {
        /*
        Debug.Log("Konvertoidaan käyttämättömät pisteet coveriksi");
```

```
        List<Unit> ownUnits = UnitManager.Instance.GetFriendlyUnitList();
            for (int i = 0; i < ownUnits.Count; i++)
            {
                Unit u = ownUnits[i];
                int ap = u.GetActionPoints();
                if (ap <= 0) continue;
                int per = u.GetCoverRegenPerUnusedAP();
                u.RegenCoverBy(ap * per);
            }
        */
    }

    public int GetTurnNumber()
    {
        return turnNumber;
    }

    public void ResetTurnNumber()
    {
        turnNumber = 1;
    }

     public void ResetTurnId()
    {
        TurnId = 0;
    }

    public bool IsPlayerTurn()
    {
        return isPlayerTurn;
    }

    public bool IsUnitsTurn(Unit unit) => unit.Team == CurrentTeam;

    public void ResetAndBegin(bool resetTurnNumber = true, bool playersPhase = true)
    {
        if (GameModeManager.SelectedMode != GameMode.SinglePlayer && Mirror.NetworkServer.active)
        {
            Debug.LogWarning("[TurnSystem] ResetAndBegin() on offline/SP-apu. Verkossa käytä NetTurnManager.ServerResetAndBegin().");
        }

        if (resetTurnNumber) turnNumber = 1;

        CurrentTeam = playersPhase ? Team.Player : Team.Enemy;
        TurnId = 0;
        var wasPlayerTurn = IsPlayerTurn();

        ForcePhase(isPlayerTurn: playersPhase, incrementTurnNumber: false);
        PlayerLocalTurnGate.Set(playersPhase);

        OnTurnStarted?.Invoke(CurrentTeam, TurnId);
    }
```

```
    public void BeginPlayersTurn(bool incrementTurnId)
    {
        if (incrementTurnId) TurnId++;
        CurrentTeam = Team.Player;
        OnTurnStarted?.Invoke(CurrentTeam, TurnId);
        ForcePhase(isPlayerTurn: true, incrementTurnNumber: false);
    }

    public void BeginEnemyTurn(bool incrementTurnId)
    {
        if (incrementTurnId) TurnId++;
        CurrentTeam = Team.Enemy;
        OnTurnStarted?.Invoke(CurrentTeam, TurnId);
        ForcePhase(isPlayerTurn: false, incrementTurnNumber: false);
    }
}
```

Assets/scripts/GameLogic/BattleLogic/WinBattle.cs

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using Mirror;

public class WinBattle : MonoBehaviour
{
    [Header("UI")]
    [SerializeField] private GameObject panel;           // koko voitto-UI:n root (piilossa aluksi)
    [SerializeField] private TextMeshProUGUI titleText;  // "Players Win!" / "Enemies Win!"
    [SerializeField] private Button playAgainButton;     // käynnistää resetin

    private bool gameEnded;

    private void Awake()
    {
        if (panel) panel.SetActive(false);
    }

    private void OnEnable()
    {
        Unit.OnAnyUnitDead    += Unit_OnAnyUnitDead;

    }

    private void OnDisable()
    {
        Unit.OnAnyUnitDead    -= Unit_OnAnyUnitDead;
    }

    private void Start()
    {
        if (panel) panel.SetActive(false);
        if (playAgainButton)
        {
            playAgainButton.onClick.RemoveAllListeners();
            playAgainButton.onClick.AddListener(OnClickPlayAgain);
        }

        // Jos aloitetaan tilasta, jossa toista puolta ei ole
       // EvaluateWin();
    }

    private void Unit_OnAnyUnitDead(object sender, System.EventArgs e)
    {
        if (GameModeManager.SelectedMode != GameMode.SinglePlayer)
        {
            if (NetMode.IsOnline) EvaluateWin_Server(); // vain server päättää
            return;
        }
```

```
        // Offline/SP
        EvaluateWin_Local();
    }

    // ---- UUSI: vain server ----
    [Server]
    private void EvaluateWin_Server()
    {
        if (gameEnded) return;
        var um = UnitManager.Instance; if (um == null) return;

        int friendCount = um.GetFriendlyUnitList().Count;
        int enemyCount = um.GetEnemyUnitList().Count;

        Debug.Log($"[WinBattle] EvaluateWin_Server: Friends={friendCount}, Enemies={enemyCount}");

        bool hostWins  = enemyCount  <= 0;
        bool hostLoses = friendCount <= 0;
        if (!(hostWins || hostLoses)) return;

        gameEnded = true; // gate, kunnes ResetService nollaa

        // Lähetä tulos jokaiselle pelaajalle henkilökohtaisesti
        foreach (var kvp in NetworkServer.connections)
        {
            var conn = kvp.Value;
            if (conn?.identity == null) continue;

            var pc = conn.identity.GetComponent<PlayerController>();
            if (!pc) continue;

            bool isHost = conn.connectionId == 0; // hostin connectionId on 0
            bool youWon = (hostWins && isHost) || (hostLoses && !isHost);
            pc.TargetShowEnd(conn, youWon); // näyttää WinBattle-paneelin clientillä
        }
    }

    // ---- Vanhasta EvaluateWinistä jää SinglePlayer-haara tähän ----
    private void EvaluateWin_Local()
    {
        if (gameEnded) return;
        var um = UnitManager.Instance; if (um == null) return;

        int friendCount = um.GetFriendlyUnitList().Count;
        int enemyCount  = um.GetEnemyUnitList().Count;

        if (enemyCount <= 0) ShowEnd("Players Win!");
        else if (friendCount <= 0) ShowEnd("Enemies Win!");
    }

    public void ShowEnd(string title)
```

```
    {
        gameEnded = true;

        if (titleText) titleText.text = title;
        if (panel) panel.SetActive(true);

    }

    private void OnClickPlayAgain()
    {
        // Yksi reitti kaikkeen: ResetService → LevelLoader
        if (NetMode.IsOnline)
        {
            ResetService.Instance.RequestReset();
            return;
        }
        gameEnded = false;
        if (panel) panel.SetActive(false);
        // OFFLINE → suoraan LevelLoaderin kautta
        LevelLoader.Instance.ReloadOffline(LevelLoader.Instance.DefaultLevel);
    }

    public void HideEndPanel()
    {
        gameEnded = false;
        if (panel) panel.SetActive(false);
    }
}
```

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("ImputManager: 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();
        }
    }

public void ForceEnablePlayerMap()
{
#if USE_NEW_INPUT_SYSTEM
    if (playerInputActions == null) playerInputActions = new PlayerInputActions();
    playerInputActions.Player.Enable();
#endif
}

public void ForceDisablePlayerMap()
```

```
{
#if USE_NEW_INPUT_SYSTEM
    playerInputActions?.Player.Disable();
#endif
}

    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 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);
    }

    [TargetRpc]
    public void TargetShowEnd(NetworkConnectionToClient conn, bool youWon)
    {
        var ui = FindFirstObjectByType<WinBattle>();
        if (ui) ui.ShowEnd(youWon ? "You win!" : "You lost");
    }
}
```

Assets/scripts/GameLogic/Player/PlayerLocalTurnGate.cs

```
using System;

/// <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 System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using Utp;

public enum GameMode { SinglePlayer, CoOp, Versus }

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


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

    private void OnEnable()
    {
        LevelLoader.LevelReady += OnLevelReady;    // ← kuuntele jokaista level-latausta
    }

    private void OnDisable()
    {
        LevelLoader.LevelReady -= OnLevelReady;
    }

     private void OnLevelReady(Scene _)
    {
        // Estää tupla spawnauksia offline-tilassa reloadin jälkeen
        if (!NetMode.IsOnline) return; // vain online-tilassa
        StartCoroutine(OfflineBootstrap()); // ← käynnistä spawnaus myös reloadin jälkeen
    }

    public bool LevelIsLoaded()
    {
        for (int i = 0; i < SceneManager.sceneCount; i++)
        {
            Scene scene = SceneManager.GetSceneAt(i);
```

```
            if (scene.isLoaded && scene.name != "Core")
            {
                return true;
            }
        }

        return false;
    }

    public Scene GetLoadedLevelScene()
    {
        for (int i = 0; i < SceneManager.sceneCount; i++)
        {
            Scene scene = SceneManager.GetSceneAt(i);
            if (scene.isLoaded && scene.name != "Core")
            {
                return scene;
            }
        }
        return default;
    }

    private IEnumerator OfflineBootstrap()
    {
        // Nämä guardit ovat jo projektissa: odota että kaikki on olemassa
        yield return new WaitUntil(() => SpawnUnitsCoordinator.Instance != null);
        yield return new WaitUntil(() => LevelGrid.Instance != null && PathFinding.Instance != null);

        if (SelectedMode == GameMode.SinglePlayer)
        {
            // Spawn offline -unitit siihen sceneen, missä koordinaattori on
            SpawnUnitsCoordinator.Instance.SpawnSinglePlayerUnits();
            LevelGrid.Instance.RebuildOccupancyFromScene();
        }
    }
}
```

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

public class DestructibleObject : NetworkBehaviour
{

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

    // Ruutujen välissä olevat seinämät eivät blokkaa. True niille jotka täyttävät kokonaisia ruutuja.
    [SerializeField] private bool blocksCell = false;

    // To prevent multiple destruction events
    private bool isDestroyed;

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

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

        // UUSI: älä tee mitään jos tämä prop on gridin ulkopuolella
        if (!LevelGrid.Instance.IsValidGridPosition(gridPosition))
        {
            Debug.LogWarning($"[DestructibleObject] {name} at {transform.position} maps OUTSIDE grid ({gridPosition}). Skip walkability.");
            return;
        }

        if (blocksCell) TryMarkBlocked();
    }

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

        // UUSI: jos PF ei ole valmis, deferoi kunnes leveys > 0
        if (PathFinding.Instance == null || PathFinding.Instance.GetWidth() <= 0)
        {
            StartCoroutine(DeferBlockUntilReady());
            return;
```

```
        }

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

    private IEnumerator DeferBlockUntilReady()
    {
        yield return new WaitUntil(() =>
            PathFinding.Instance != null &&
            PathFinding.Instance.GetWidth() > 0);

        if (!LevelGrid.Instance.IsValidGridPosition(gridPosition)) yield break;

        PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, false);
        _walkabilitySet = true;
    }

    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));
            PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, true);
            EdgeBaker.Instance.RebakeEdgesAround(gridPosition);
        }
    }

    private void PlayDestroyFx(Vector3 hitPosition, int overkill)
    {

        // Sama sijainti, sama rotaatio, sama parent kuin alkuperäisellä
        var go = SpawnRouter.SpawnLocal(
            prefab: objectDestroyPrefab,
            pos: transform.position,
            rot: transform.rotation,
            source: transform,              // ⟵ ohjaa samaan sceneen kuin tuhottava objekti
            sceneName: LevelLoader.Instance.CurrentLevel,
            parent: null,                   // ⟵ älä periytä Core-parenttia vahingossa
            beforeReturn: go =>
            {

                // jos haluat säilyttää skaalan:
                go.transform.localScale = transform.localScale;
                ApplyPushForceToChildren(go.transform, 10f * overkill, hitPosition, 10f);
            }
        );
    }

    [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)
        {
            // Server: vapauta ruutu ja rebake serverillä
            PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, true);
            EdgeBaker.Instance.RebakeEdgesAround(gridPosition);

            // Lähetä sama clienteille ennen tuhoa
            RpcOnDestroyed(gridPosition);

            // Pieni hengähdys (valinnainen, usein ei pakollinen)
            // yield return null;

            NetworkServer.Destroy(gameObject);
        } else {
            // Offline-tapaus tms.
            Destroy(gameObject);
        }
    }

    // Lisää tämä luokkaan
    [ClientRpc]
    private void RpcOnDestroyed(GridPosition pos)
    {

        var lg = LevelGrid.Instance;
        var pf = PathFinding.Instance;
        var eb = EdgeBaker.Instance;

        if (lg != null && pf != null)
            pf.SetIsWalkableGridPosition(pos, true);

        if (lg != null && pf != null && eb != null)
            eb.RebakeEdgesAround(pos);
    }

    // Varmistus myös tilanteeseen, jossa RPC hukkuu tai tulee myöhässä
    public override void OnStopClient()
    {
```

```
        var lg = LevelGrid.Instance;
        var pf = PathFinding.Instance;
        var eb = EdgeBaker.Instance;

        // Palauta walkable vain jos LevelGrid + PathFinding ovat olemassa
        if (lg != null && pf != null)
            pf.SetIsWalkableGridPosition(gridPosition, true);

        // Älä rebakea jos yksikin puuttuu (teardownissa usein puuttuu)
        if (lg != null && pf != null && eb != null)
            eb.RebakeEdgesAround(gridPosition);
    }

    [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;

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/LosBlocker.cs

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

[DisallowMultipleComponent]
public class LoSBlocker : MonoBehaviour
{
    public enum Mode { Manual, AutoFromBounds }

    [Header("Mode")]
    [SerializeField] private Mode mode = Mode.AutoFromBounds;

    [Tooltip("Jos Manual: true = blokkaa LoS:n")]
    [SerializeField] private bool blocksLineOfSight = true;

    [Header("AutoFromBounds-asetukset")]
    [Tooltip("Yli tämän y-kynnyksen (ruudun pohjasta) tulkitaan 'korkeaksi'")]
    [SerializeField] private float heightThresholdY = 1.7f;

    [Tooltip("Käytä näitä kollidereita. Tyhjä = hae automaattisesti lapsista.")]
    [SerializeField] private List<Collider> colliders = new();

    // Seurataan mitä ruutuja tämä tällä hetkellä kattaa
    private readonly HashSet<GridPosition> _covered = new();
    private bool _registered;
    private LevelGrid _lg;

    void OnEnable()
    {
        _lg = LevelGrid.Instance;
        if (colliders.Count == 0)
            colliders.AddRange(GetComponentsInChildren<Collider>());

        Rebuild();  // laske ruudut + blokkaako
        Register(); // vie registryyn
    }

    void OnDisable()
    {
        Unregister();
        colliders.Clear();
        _covered.Clear();
    }

    /// <summary>
    /// Kutsu tätä kun objektin korkeus/muoto muuttuu (esim. tuhoutuessa).
    /// </summary>
    public void Rebuild()
    {
        if (_lg == null) return;
```

```
        // 1) Laske yhdistetty bounds (vain colliderit jotka ovat aktiivisia)
        var haveAny = false;
        var bounds = new Bounds(transform.position, Vector3.zero);
        foreach (var c in colliders)
        {
            if (c == null || !c.enabled) continue;
            if (!haveAny) { bounds = c.bounds; haveAny = true; }
            else          { bounds.Encapsulate(c.bounds); }
        }
        if (!haveAny)
        {
            blocksLineOfSight = false;    // ei kollidereita → ei blokkaa
            RefreshCoveredTiles(bounds, hasBounds:false);
            return;
        }

        // 2) Määritä blokkaako: Manual vs Auto
        if (mode == Mode.AutoFromBounds)
        {
            // Ruudun pohjataso: mittaamme korkeuden paikallisesti jokaisessa ruudussa,
            // mutta yksinkertaisuuden vuoksi arvioi koko objektin "max korkeus"
            float maxTop = bounds.max.y;

            // Arvioi peruslattia käyttämällä bounds.min ja LevelGridiä
            var gpMin = _lg.GetGridPosition(bounds.min);
            var basePos = _lg.GetWorldPosition(gpMin);
            float baseY = basePos.y;

            float topAboveBase = maxTop - baseY;
            blocksLineOfSight = topAboveBase >= heightThresholdY;
        }
        // Manual: blocksLineOfSight pysyy sellaisenaan

        // 3) Päivitä mitkä ruudut tämä kattaa
        RefreshCoveredTiles(bounds, hasBounds:true);
    }

    /// <summary>
    /// Rekisteröi/poistaa rekisteristä nykyiset ruudut.
    /// </summary>
    private void Register()
    {
        if (_registered || !blocksLineOfSight || _covered.Count == 0) return;
        LoSBlockerRegistry.AddTiles(_covered);
        _registered = true;
    }

    private void Unregister()
    {
        if (!_registered) return;
        LoSBlockerRegistry.RemoveTiles(_covered);
        _registered = false;
```

```
    }

    /// <summary>
    /// Päivittää _covered-ruudut boundsien perusteella ja synkkaa registryyn.
    /// </summary>
    private void RefreshCoveredTiles(Bounds worldBounds, bool hasBounds)
    {
        // Poista vanhat ruudut registryltä
        Unregister();
        _covered.Clear();

        if (!hasBounds || _lg == null || !blocksLineOfSight) return;

        // Muunna bounds → grid-alue
        var min = _lg.GetGridPosition(worldBounds.min);
        var max = _lg.GetGridPosition(worldBounds.max);

        // Jos objekti voi levitä usealle floorille, tässä voi rajata/vahvistaa.
        int floor = min.floor;

        for (int x = Mathf.Min(min.x, max.x); x <= Mathf.Max(min.x, max.x); x++)
        for (int z = Mathf.Min(min.z, max.z); z <= Mathf.Max(min.z, max.z); z++)
        {
            var gp = new GridPosition(x, z, floor);
            if (_lg.IsValidGridPosition(gp))
                _covered.Add(gp);
        }

        // Rekisteröi päivitetty setti
        Register();
    }
}
```

Assets/scripts/GameObjects/ObjectSpawnPlaceHolder.cs

```
using Mirror;
using UnityEngine;
using UnityEngine.SceneManagement;

public class ObjectSpawnPlaceHolder : MonoBehaviour
{
    [SerializeField] private GameObject objectPrefab;
    public GameObject Prefab => objectPrefab;

    private void Start()
    {
        // OFFLINE: käytä SpawnRouter.SpawnLocal → asettaa instanssin samaan level-sceneen kuin placeholder
        if (!NetworkClient.active && !NetworkServer.active)
        {
            SpawnRouter.SpawnLocal(
                prefab: objectPrefab,
                pos: transform.position,
                rot: transform.rotation,
                source: transform,
                sceneName: gameObject.scene.name
            );
            Destroy(gameObject);
            return;
        }

        // PUHDAS CLIENT: server spawnaa → placeholder pois
        if (NetworkClient.active && !NetworkServer.active)
        {
            Destroy(gameObject);
        }
    }

    // Kutsutaan serveriltä (esim. MapContentSpawnerista)
    public GameObject CreteObject()
    {
        if (!NetworkServer.active) return null;

        Scene levelScene = gameObject.scene;

        var go = SpawnRouter.SpawnNetworkServer(
            prefab: objectPrefab,
            pos: transform.position,
            rot: transform.rotation,
            source: transform,            // → tämän placeholderin scene
            sceneName: levelScene.name,   // → ohita Core, lukitse oikeaan level-sceneen
            parent: null,
            owner: null,                  // ympäristöpropsit eivät tarvitse owneria
            beforeSpawn: obj =>
            {
                // Jos haluat inittejä (HP tms.), tee ne tässä
```

```
                // esim: if (obj.TryGetComponent<DestructibleObject>(out var d)) d.Init(...);
            }
        );

        Destroy(gameObject);
        return go;
    }
}
```

Assets/scripts/Grid/GridObject.cs

```
using System.Collections.Generic;

// <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);
        */

        if (unit == null) return;
        var list = GetUnitList();
        if (!list.Contains(unit)) list.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()
    {
        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;

// <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 UnityEngine;
using Mirror;
using System.Collections;

[DefaultExecutionOrder(-100)]
public class GridSystemVisual : MonoBehaviour
{
    public static GridSystemVisual Instance { get; private set; }

    [Header("Mouse-plane filter")]
    [SerializeField] private bool filterToMousePlanes = true;

    [Header("Team Vision Overlay")]
    [SerializeField] private bool teamVisionEnabled = true;
    [SerializeField] private GridVisualType teamVisionType = GridVisualType.Yellow;

    [SerializeField] private bool invertTeamVision = true;
    [SerializeField] private GridVisualType fogType = GridVisualType.TeamVision; // laita tähän harmaa materiaali

    private readonly HashSet<GridPosition> _lastActionCells = new();
    private readonly List<GridPosition> _tmpList = new(256);

    float delayTime;

    private bool _isReady = false;

    [Serializable]
    public struct GridVisualTypeMaterial
    {
        public GridVisualType gridVisualType;
        public Material material;
    }

    public enum GridVisualType
    {
        white,
        Blue,
        Red,
        RedSoft,
        Yellow,
        TeamVision,
        UnitPersonalVision,
        UnitOverwatchVision

    }

    [SerializeField] private Transform gridSystemVisualSinglePrefab;
    [SerializeField] private List<GridVisualTypeMaterial> gridVisualTypeMaterialList;
```

```
    private GridSystemVisualSingle[,,] gridSystemVisualSingleArray;

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

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

        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 t = Instantiate(gridSystemVisualSinglePrefab, LevelGrid.Instance.GetWorldPosition(gridPosition), Quaternion.identity);
                    gridSystemVisualSingleArray[x, z, floor] = t.GetComponent<GridSystemVisualSingle>();
                }

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

        if (TeamVisionService.Instance != null)
            TeamVisionService.Instance.OnTeamVisionChanged += HandleTeamVisionChanged;

        if (NetworkClient.active && TeamVisionService.Instance != null)
        {
            int myTeam = GetLocalPlayerTeamId();
            TeamVisionService.Instance.ClearTeamVision(myTeam);
        }

        _isReady = true;
        UpdateGridVisuals();
    }

    private void OnDisable()
    {
        UnitActionSystem.Instance.OnSelectedActionChanged -= UnitActionSystem_OnSelectedActionChanged;
```

```
        UnitActionSystem.Instance.OnBusyChanged -= UnitActionSystem_OnBusyChanged;
        LevelGrid.Instance.onAnyUnitMoveGridPosition -= LevelGrid_onAnyUnitMoveGridPosition;

        if (TeamVisionService.Instance != null)
            TeamVisionService.Instance.OnTeamVisionChanged -= HandleTeamVisionChanged;
    }

    private bool HasMousePlaneAt(in GridPosition gp)
        => !filterToMousePlanes || (MousePlaneMap.Instance && MousePlaneMap.Instance.Has(gp));

    public int GetLocalPlayerTeamId()
    {
        GameMode mode = GameModeManager.SelectedMode;

        if (mode == GameMode.SinglePlayer || mode == GameMode.CoOp)
            return 0;

        if (mode == GameMode.Versus)
        {
            if (NetworkServer.active && !NetworkClient.active)
            {
                Debug.LogWarning("[GridSystemVisual] Running on dedicated server - no local player team");
                return 0;
            }

            if (NetworkClient.localPlayer != null)
            {
                var localPlayerUnit = NetworkClient.localPlayer.GetComponent<Unit>();
                if (localPlayerUnit != null)
                {
                    bool isHost = NetworkServer.active;
                    int teamId = isHost ? 0 : 1;
                    return teamId;
                }
            }

            bool fallbackIsHost = NetworkServer.active;
            int fallbackTeam = fallbackIsHost ? 0 : 1;
            return fallbackTeam;
        }

        return 0;
    }

    public void HideAllGridPositions()
    {
        if (!_isReady || gridSystemVisualSingleArray == null) return;
        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();
    }
```

```
public void ShowGridPositionList(List<GridPosition> gridPositionList, GridVisualType gridVisualType)
{
    var mat = GetGridVisualTypeMaterial(gridVisualType);
    foreach (var gp in gridPositionList)
    {
        if (filterToMousePlanes && !HasMousePlaneAt(gp)) continue;
        var cell = gridSystemVisualSingleArray[gp.x, gp.z, gp.floor];
        if (cell != null) cell.Show(mat);
    }
}

public void UpdateGridVisuals()
{
    if (!_isReady || gridSystemVisualSingleArray == null) return;

    HideAllGridPositions();
    RedrawPersistentOverwatch();
    _lastActionCells.Clear();

    if (teamVisionEnabled && TeamVisionService.Instance != null)
    {
        if (invertTeamVision)
        {
            DrawTeamVisionOverlayInverted();
        }
        else
        {
            DrawTeamVisionOverlay(); // nykyinen
        }
    }

    Unit selectedUnit = UnitActionSystem.Instance.GetSelectedUnit();
    if (selectedUnit == null) return;

    BaseAction selectedAction = UnitActionSystem.Instance.GetSelectedAction();

    GridVisualType gridVisualType;

    switch (selectedAction)
    {
        default:
        case MoveAction:
            gridVisualType = GridVisualType.white;
            break;

        case OverwatchAction:

            selectedAction.GetUnit().GetComponent<UnitVision>().ShowUnitPersonalVision();
            gridVisualType = GridVisualType.white;
            break;
```

```
            case ShootAction shoot:
                {
                    gridVisualType = GridVisualType.Red;

                    var origin = selectedUnit.GetGridPosition();
                    int range = shoot.GetMaxShootDistance();

                    var cfg = LoSConfig.Instance;
                    var visible = RaycastVisibility.ComputeVisibleTilesRaycast(
                        origin, range,
                        cfg.losBlockersMask, cfg.eyeHeight, cfg.samplesPerCell, cfg.insetWU
                    );
                    visible.RemoveWhere(gp => !RaycastVisibility.HasLineOfSightRaycastHeightAware(
                        origin, gp, cfg.losBlockersMask, cfg.eyeHeight, cfg.samplesPerCell, cfg.insetWU));

                    _tmpList.Clear();
                    _tmpList.AddRange(visible);
                    ShowAndMark(_tmpList, GridVisualType.RedSoft);
                    break;
                }

            case GranadeAction:
                gridVisualType = GridVisualType.Yellow;
                break;

            case MeleeAction:
                gridVisualType = GridVisualType.Red;
                ShowAndMark(BuildRangeSquare(selectedUnit.GetGridPosition(), 1), GridVisualType.RedSoft);
                break;

            case InteractAction:
                gridVisualType = GridVisualType.Yellow;
                break;
        }

        ShowAndMark(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 m in gridVisualTypeMaterialList)
            if (m.gridVisualType == gridVisualType)
                return m.material;
```

```
        Debug.LogError("Could not find GridVisualTypeMaterial for GridVisualType " + gridVisualType);
        return null;
    }

    private void HandleTeamVisionChanged(int teamId)
    {
        if (!teamVisionEnabled) return;

        int myTeam = GetLocalPlayerTeamId();
        if (teamId != myTeam)
        {
            return;
        }

        UpdateGridVisuals();
    }

    private void DrawTeamVisionOverlay()
    {
        if (TeamVisionService.Instance == null) return;

        int myTeam = GetLocalPlayerTeamId();
        var snap = TeamVisionService.Instance.GetVisibleTilesSnapshot(myTeam);
        if (snap == null) return;

        _tmpList.Clear();
        foreach (var gp in snap)
            if (!_lastActionCells.Contains(gp) && (!filterToMousePlanes || HasMousePlaneAt(gp)))
                _tmpList.Add(gp);

        ShowGridPositionList(_tmpList, teamVisionType);
    }

    private void DrawTeamVisionOverlayInverted()
    {
        if (TeamVisionService.Instance == null) return;

        int myTeam = GetLocalPlayerTeamId();
        var snap = TeamVisionService.Instance.GetVisibleTilesSnapshot(myTeam);
        if (snap == null) return;

        // O(1) tarkistukset
        var visible = new HashSet<GridPosition>(snap);

        var mat = GetGridVisualTypeMaterial(fogType);
        int W = LevelGrid.Instance.GetWidth();
        int H = LevelGrid.Instance.GetHeight();
        int F = LevelGrid.Instance.GetFloorAmount();

        for (int x = 0; x < W; x++)
            for (int z = 0; z < H; z++)
```

```
            for (int f = 0; f < F; f++)
            {
                var gp = new GridPosition(x, z, f);

                // Sama suodatus kuin muuallakin (MousePlaneMap), ettei harmaa leviä “väärään kerrokseen”
                if (filterToMousePlanes && !(MousePlaneMap.Instance && MousePlaneMap.Instance.Has(gp))) continue;

                // Älä peitä tuoreita action-ruutuja
                if (_lastActionCells.Contains(gp)) continue;

                // Näkyvät → jätetään puhtaiksi; näkymättömät → harmaa
                if (!visible.Contains(gp))
                {
                    var cell = gridSystemVisualSingleArray[x, z, f];
                    if (cell != null) cell.Show(mat);
                }
            }
    }

    private void ShowAndMark(IEnumerable<GridPosition> cells, GridVisualType type)
    {
        var mat = GetGridVisualTypeMaterial(type);
        foreach (var gp in cells)
        {
            if (filterToMousePlanes && !HasMousePlaneAt(gp)) continue;
            var cell = gridSystemVisualSingleArray[gp.x, gp.z, gp.floor];
            if (cell != null) cell.Show(mat);
            _lastActionCells.Add(gp);
        }
    }

    private List<GridPosition> BuildRangeSquare(GridPosition center, int range)
    {
        var list = new List<GridPosition>();
        for (int x = -range; x <= range; x++)
            for (int z = -range; z <= range; z++)
            {
                var gp = center + new GridPosition(x, z, 0);
                if (LevelGrid.Instance.IsValidGridPosition(gp))
                    list.Add(gp);
            }
        return list;
    }

    private readonly Dictionary<Unit, List<GridPosition>> _owPersistent = new();

    private void RedrawPersistentOverwatch()
    {
        foreach (var kv in _owPersistent)
            ShowGridPositionList(kv.Value, GridVisualType.UnitOverwatchVision);
    }
```

```
    public void ClearTeamOverwatchVisuals(int teamId)
    {
        var keysToRemove = new List<Unit>();

        foreach (var kv in _owPersistent)
        {
            if (kv.Key != null && kv.Key.GetTeamID() == teamId)
            {
                keysToRemove.Add(kv.Key);
            }
        }

        foreach (var unit in keysToRemove)
        {
            // Käytä RemovePersistentOverwatch jotta ajastimet peruuntuvat
            RemovePersistentOverwatch(unit);
        }
    }

//  [Header("Enemy Overwatch Display")]
//  [SerializeField] private float enemyOverwatchVisibilityDuration = 3f;

    private readonly Dictionary<Unit, Coroutine> _owTimers = new();

    public void AddPersistentOverwatch(Unit u, List<GridPosition> tiles)
    {
        if (u == null) return;

        _owPersistent[u] = tiles;
        RedrawPersistentOverwatch();

        // Jos tämä on vihollisen yksikkö (ei oma tiimi), käynnistä ajastin
        int myTeam = GetLocalPlayerTeamId();
        if (u.GetTeamID() != myTeam)
        {
            StartEnemyOverwatchTimer(u);
        }
    }

    private void StartEnemyOverwatchTimer(Unit enemyUnit)
    {
        // Peruuta vanha ajastin jos on käynnissä
        if (_owTimers.TryGetValue(enemyUnit, out var oldCoroutine))
        {
            if (oldCoroutine != null)
                StopCoroutine(oldCoroutine);
        }

        // Käynnistä uusi ajastin
        var coroutine = StartCoroutine(Co_HideEnemyOverwatchAfterDelay(enemyUnit));
        _owTimers[enemyUnit] = coroutine;
    }
```

```
    private IEnumerator Co_HideEnemyOverwatchAfterDelay(Unit enemyUnit)
    {
        yield return new WaitForSeconds(delayTime);

        // Piilota tämä vihollisen overwatch
        RemovePersistentOverwatch(enemyUnit);
        _owTimers.Remove(enemyUnit);

        if (_isReady)
        {
            UpdateGridVisuals();
        }
    }

    public void RemovePersistentOverwatch(Unit u)
    {
        // Peruuta ajastin jos on käynnissä
        if (_owTimers.TryGetValue(u, out var coroutine))
        {
            if (coroutine != null)
                StopCoroutine(coroutine);
            _owTimers.Remove(u);
        }

        _owPersistent.Remove(u);

        if (_isReady)
        {
            UpdateGridVisuals();
        }
    }

    public void ClearAllPersistentOverwatch()
    {
        // Peruuta kaikki ajastimet
        foreach (var kv in _owTimers)
        {
            if (kv.Value != null)
                StopCoroutine(kv.Value);
        }
        _owTimers.Clear();

        _owPersistent.Clear();
        if (_isReady) UpdateGridVisuals();
    }
}
```

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()
    {
        if (meshRenderer != null)
            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;
private const float CELLSIZE = 2f;
[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/LoSBlockerRegistry.cs

```
using System.Collections.Generic;

public static class LoSBlockerRegistry
{
    // Kuinka monella "tall-blockerilla" ruutu on peitetty
    private static readonly Dictionary<GridPosition, int> _counts = new();

    public static void Reset() => _counts.Clear();

    public static void AddTiles(IEnumerable<GridPosition> tiles)
    {
        foreach (var t in tiles)
        {
            _counts.TryGetValue(t, out int c);
            _counts[t] = c + 1;
        }
    }

    public static void RemoveTiles(IEnumerable<GridPosition> tiles)
    {
        foreach (var t in tiles)
        {
            if (!_counts.TryGetValue(t, out int c)) continue;
            c--;
            if (c <= 0) _counts.Remove(t);
            else _counts[t] = c;
        }
    }

    public static bool TileHasTallBlocker(GridPosition p)
        => _counts.TryGetValue(p, out int c) && c > 0;
}
```

Assets/scripts/Grid/MousePlaneMap.cs

```
using System.Collections;
using UnityEngine;

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

    [Header("Scan settings")]
    [SerializeField] private LayerMask mousePlaneMask;    // "MousePlane"
    [SerializeField] private float cellSizeWU = 2f;       // sama kuin LevelGrid
    [SerializeField] private float halfThicknessWU = 0.3f;
    [SerializeField] private float yOffsetWU = 0.02f;

    private BitArray bits;
    private int W, H, F;

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

    public void Rebuild()
    {
        var lg = LevelGrid.Instance;
        if (!lg) return;

        W = lg.GetWidth(); H = lg.GetHeight(); F = lg.GetFloorAmount();
        bits = new BitArray(W * H * F, false);

        var half = new Vector3(cellSizeWU * 0.5f, halfThicknessWU, cellSizeWU * 0.5f);

        for (int f = 0; f < F; f++)
        for (int z = 0; z < H; z++)
        for (int x = 0; x < W; x++)
        {
            var gp = new GridPosition(x, z, f);
            var c  = lg.GetWorldPosition(gp) + Vector3.up * yOffsetWU;
            bool hasPlane = Physics.CheckBox(c, half, Quaternion.identity, mousePlaneMask, QueryTriggerInteraction.Collide);
            bits[Idx(gp)] = hasPlane;
        }
    }

    public bool Has(in GridPosition gp)
    {
        if (bits == null) return false;
        if (gp.x < 0 || gp.x >= W || gp.z < 0 || gp.z >= H || gp.floor < 0 || gp.floor >= F) return false;
        return bits[Idx(gp)];
    }
```

```
    // Valmius tulevaisuuteen (esim. räjähdys tekee reiän):
    public bool Remove(in GridPosition gp) { if (!Has(gp)) return false; bits[Idx(gp)] = false; return true; }
    public bool Add(in GridPosition gp)    { if (Has(gp))  return false; bits[Idx(gp)] = true;  return true; }

    private int Idx(in GridPosition gp) => gp.floor * (W * H) + gp.z * W + gp.x;
}
```

Assets/scripts/Grid/VisibilityService.cs

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

public static class VisibilityService
{
    // Toleranssi
    private const float EPS = 1e-4f;

    // Välimuisti ruudun "onko korkea blokkeri" -tiedolle
    // Tyhjennä esim. vuoron vaihtuessa tai kun kenttä muuttuu
    private static readonly Dictionary<GridPosition, bool> _tallBlockerCache = new();

    /// <summary>Tyhjennä korkeablokkeri-välimuisti (kutsu esim. vuoron vaihtuessa, kun yksiköt/liikuteltavat esteet liikkuvat tai kun map spawnaa asioita).</summary>
    public static void ResetTallBlockerCache() => _tallBlockerCache.Clear();

    /// <summary>
    /// Palauttaa näkyvät ruudut (sama floor) originista maxRangeen.
    /// Estäjät: 1) koko-ruudun korkeat esteet (PF: !walkable), 2) välissä seisovat unitit.
    /// Ei käytä EdgeBakereita tässä vaiheessa.
    /// </summary>
    public static HashSet<GridPosition> ComputeVisibleTiles(GridPosition origin, int maxRange, bool occludeByUnits = true)
    {
        var visible = new HashSet<GridPosition>();
        var lg = LevelGrid.Instance;
        var pf = PathFinding.Instance;
        if (lg == null || pf == null) return visible;

        for (int dx = -maxRange; dx <= maxRange; dx++)
        {
            for (int dz = -maxRange; dz <= maxRange; dz++)
            {
                var cost = SircleCalculator.Sircle(dx, dz);
                if (cost > 10 * maxRange) continue;

                var gp = new GridPosition(origin.x + dx, origin.z + dz, origin.floor);
                if (!lg.IsValidGridPosition(gp)) continue;

                if (HasLineOfSight(origin, gp, occludeByUnits))
                    visible.Add(gp);
            }
        }
        return visible;
    }

    /// <summary>
    /// Bresenham 2D viiva ruutujen (x,z) yli; floor säilyy samana.
    /// </summary>
    private static IEnumerable<GridPosition> Line(GridPosition from, GridPosition to)
    {
        int x0 = from.x, z0 = from.z;
```

```
        int x1 = to.x, z1 = to.z;

        int dx = Mathf.Abs(x1 - x0);
        int dz = Mathf.Abs(z1 - z0);
        int sx = x0 < x1 ? 1 : -1;
        int sz = z0 < z1 ? 1 : -1;
        int err = dx - dz;

        // mukaan myös lähtö
        yield return from;

        while (x0 != x1 || z0 != z1)
        {
            int e2 = 2 * err;
            if (e2 > -dz) { err -= dz; x0 += sx; }
            if (e2 < dx)  { err += dx; z0 += sz; }
            yield return new GridPosition(x0, z0, from.floor);
        }
    }

#if UNITY_EDITOR
    private static void DebugCheckEdgeSymmetry(GridPosition prev, int dx, int dz)
    {
        // Tarkistetaan vain diagonaaliaskeleella
        if (dx == 0 || dz == 0) return;

        var horiz = dx > 0 ? EdgeMask.E : EdgeMask.W;
        var mid   = new GridPosition(prev.x + dx, prev.z, prev.floor);
        var vert  = dz > 0 ? EdgeMask.N : EdgeMask.S;

        // Vastareunat naapurisoluista
        var neighborH = new GridPosition(prev.x + dx, prev.z, prev.floor);
        var oppH      = dx > 0 ? EdgeMask.W : EdgeMask.E;

        var neighborV = new GridPosition(mid.x, mid.z + dz, mid.floor);
        var oppV      = dz > 0 ? EdgeMask.S : EdgeMask.N;

        bool hA = EdgeOcclusion.HasTallWall(prev,      horiz);
        bool hB = EdgeOcclusion.HasTallWall(neighborH, oppH);
        bool vA = EdgeOcclusion.HasTallWall(mid,       vert);
        bool vB = EdgeOcclusion.HasTallWall(neighborV, oppV);

        if (hA != hB || vA != vB)
        {
            Debug.LogWarning(
                $"[LoS] EdgeOcclusion ei näytä olevan symmetrinen diagonaaliaskeleella.\n" +
                $"  H: ({prev.x},{prev.z},f{prev.floor}).{horiz}={hA}  vs  " +
                $"({neighborH.x},{neighborH.z},f{neighborH.floor}).{oppH}={hB}\n" +
                $"  V: ({mid.x},{mid.z},f{mid.floor}).{vert}={vA}  vs  " +
                $"({neighborV.x},{neighborV.z},f{neighborV.floor}).{oppV}={vB}"
            );
        }
```

```
    }
#endif

    public static bool HasLineOfSight(GridPosition from, GridPosition to, bool occludeByUnits = true)
    {
        if (from.floor != to.floor) return false;

        var lg = LevelGrid.Instance;
        if (lg == null) return false;

        // Early-out: LoS itseensä
        if (from.Equals(to)) return true;

        bool first = true;
        GridPosition prev = default;

        foreach (var p in Line(from, to))
        {
            if (first) { first = false; prev = p; continue; }

            // 1) Kokoruutu-korkeat esteet blokkaavat "pehmeästi" kuten ennenkin
            if (LoSBlockerRegistry.TileHasTallBlocker(p))
            {
#if UNITY_EDITOR
                Debug.Log($"[LoS] Blokattu FULL-TILE esteestä ruudussa ({p.x},{p.z},f{p.floor})  reitillä ({from.x},{from.z})->({to.x},{to.z})");
#endif
                return false;
            }

            // 2) Ohuet seinät: tarkista Bresenham-askeleen ylittämä(t) reuna(t)
            int dx = p.x - prev.x;
            int dz = p.z - prev.z;

#if UNITY_EDITOR
            // Tarkista bake-symmetria diagonaaliaskeleella (vain editorissa)
            DebugCheckEdgeSymmetry(prev, dx, dz);
#endif

            if (dx != 0)
            {
                var horiz = dx > 0 ? EdgeMask.E : EdgeMask.W;
                if (EdgeOcclusion.HasTallWall(prev, horiz))
                {
#if UNITY_EDITOR
                    Debug.Log($"[LoS] Blokattu OHUEN seinän HORIZ reuna: prev=({prev.x},{prev.z},f{prev.floor}) edge={horiz}  reitillä ({from.x},{from.z})->({to.x},{to.z})");
#endif
                    return false;
                }

                // Diagonaalisteppi: ylitetään myös pystyreuna "väliruudusta"
                if (dz != 0)
                {
```

```
                    var mid = new GridPosition(prev.x + dx, prev.z, prev.floor);
                    var vert = dz > 0 ? EdgeMask.N : EdgeMask.S;
                    if (EdgeOcclusion.HasTallWall(mid, vert))
                    {
#if UNITY_EDITOR
                        Debug.Log($"[LoS] Blokattu OHUEN seinän VERT reuna: mid=({mid.x},{mid.z},f{mid.floor}) edge={vert}  reitillä ({from.x},{from.z})->({to.x},{to.z})");
#endif
                        return false;
                    }
                }
            }
            else if (dz != 0)
            {
                var vert = dz > 0 ? EdgeMask.N : EdgeMask.S;
                if (EdgeOcclusion.HasTallWall(prev, vert))
                {
#if UNITY_EDITOR
                    Debug.Log($"[LoS] Blokattu OHUEN seinän VERT reuna: prev=({prev.x},{prev.z},f{prev.floor}) edge={vert}  reitillä ({from.x},{from.z})->({to.x},{to.z})");
#endif
                    return false;
                }
            }

            // 3) Välissä seisovat unitit voivat edelleen blokata (valinnainen)
            if (occludeByUnits && !p.Equals(to) && lg.HasAnyUnitOnGridPosition(p))
            {
#if UNITY_EDITOR
                Debug.Log($"[LoS] Blokattu UNITIN vuoksi ruudussa ({p.x},{p.z},f{p.floor})  reitillä ({from.x},{from.z})->({to.x},{to.z})");
#endif
                return false;
            }

            prev = p;
        }

        return true;
    }
}
```

Assets/scripts/Helpers/AllUnitsList.cs

```
using Mirror;
using UnityEngine;
/// <summary>
/// Only used for cleaning the field from units.
/// </summary>
[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/BalanceDB.cs

```
using UnityEngine;

public static class BalanceDB {
    const string RES_PATH = "CombatRanges"; // => Assets/Resources/CombatRanges.asset
    static CombatRanges _cached;
    static bool _warned;

    public static CombatRanges Ranges {
        get {
            if (GameBalance.R != null) return GameBalance.R;              // ensisijainen
            if (_cached != null) return _cached;                           // cache
            _cached = UnityEngine.Resources.Load<CombatRanges>(RES_PATH);  // fallback
            if (_cached == null && !_warned) {
                _warned = true;
                Debug.LogWarning("[BalanceDB] CombatRanges puuttuu.\n"+
                                 "- Suositus: aseta se Core→GameBalance.ranges -kenttään.\n"+
                                 "- Fallback: laita asset polkuun Assets/Resources/"+RES_PATH+".asset");
            }
            return _cached;
        }
    }
}
```

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);

        UnitManager.Instance.ClearAllUnitLists();
        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/DebrisUtil.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

/// <summary>
/// Kevyt util: poista Debris-layerillä olevat paikalliset GameObjectit kaikista ei-Core -sceneistä.
/// Tämä on pelkkä “varmistuskerros” – varsinainen siivous tapahtuu jo scene-unloadissa.
/// </summary>
public static class DebrisUtil
{
    public static int DestroyAllDebrisExceptCore(string coreName = "Core", string debrisLayerName = "Debris")
    {
        int debrisLayer = LayerMask.NameToLayer(debrisLayerName);
        if (debrisLayer == -1) return 0;

        int destroyed = 0;
        for (int i = 0; i < SceneManager.sceneCount; i++)
        {
            var s = SceneManager.GetSceneAt(i);
            if (!s.isLoaded || s.name == coreName) continue;

            var roots = s.GetRootGameObjects();
            for (int r = 0; r < roots.Length; r++)
            {
                destroyed += DestroyDebrisRecursive(roots[r].transform, debrisLayer);
            }
        }
        return destroyed;
    }

    private static int DestroyDebrisRecursive(Transform t, int debrisLayer)
    {
        int cnt = 0;
        for (int i = t.childCount - 1; i >= 0; i--)
        {
            var c = t.GetChild(i);
            if (c.gameObject.layer == debrisLayer)
            {
                Object.Destroy(c.gameObject);
                cnt++;
            }
            else
            {
                cnt += DestroyDebrisRecursive(c, debrisLayer);
            }
        }
        return cnt;
    }
}
```

Assets/scripts/LevelCreation/LevelCatalog.cs

```
// RogueShooter/LevelCatalog.cs
using System;
using System.Collections.Generic;
using UnityEngine;

#if UNITY_EDITOR
using UnityEditor;
#endif


[CreateAssetMenu(fileName = "LevelCatalog", menuName = "RogueShooter/Level Catalog")]
public class LevelCatalog : ScriptableObject {
    public List<LevelEntry> levels = new();

    public int Count => levels?.Count ?? 0;
    public LevelEntry Get(int i) => (i >= 0 && i < Count) ? levels[i] : null;
    public int IndexOfScene(string sceneName) => levels.FindIndex(l => l != null && l.sceneName == sceneName);

    private void OnValidate()
    {
#if UNITY_EDITOR
        foreach (var l in levels)
        {
            if (l == null || string.IsNullOrEmpty(l.sceneName)) continue;
            bool inBuild = false;
            foreach (var s in EditorBuildSettings.scenes)
            {
                if (s.enabled && s.path.EndsWith($"{l.sceneName}.unity")) { inBuild = true; break; }
            }
            if (!inBuild)
                Debug.LogWarning($"[LevelCatalog] '{l.sceneName}' ei ole Build Settingsissä (enabled).");
        }
#endif
    }

}

[Serializable]
public class LevelEntry
{
    [Tooltip("Scene name täsmälleen Build Settingsissä")]
    public string sceneName;
    public string displayName;
    public Sprite thumbnail;

    // Valinnainen metadata editorityöhön
    public Vector3Int gridSize = new(30, 1, 30);
    public int floors = 1;
}
```

Assets/scripts/LevelCreation/LevelLoader.cs

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

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

    [SerializeField] private string coreSceneName = "Core";
    [SerializeField] private string defaultLevel = "Level 0";

    public string CoreSceneName => coreSceneName;
    public string DefaultLevel => defaultLevel;
    public string CurrentLevel { get; private set; }

    // [SerializeField] private bool forceDefaultOnStart = true;

    // --- NÄMÄ KAKSI UUTTA ---
    public static bool IsServerLevelReady { get; private set; }
    public static void SetServerLevelReady(bool ready) => IsServerLevelReady = ready;

    // Event pysyy LevelLoaderissa; muiden pitää kutsua RaiseLevelReady(...)
    public static event Action<Scene> LevelReady;
    public static void RaiseLevelReady(Scene scene) => LevelReady?.Invoke(scene);

    [SerializeField] private LevelCatalog catalog;
    [SerializeField] private int currentIndex;

#if UNITY_EDITOR
    private const string EDITOR_REQ_KEY = "RS_EditorRequestedLevel";
#endif

    private void Awake()
    {

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

#if UNITY_EDITOR

        // 1) Lue editorin one-shot-pyyntö
        string req = PlayerPrefs.GetString(EDITOR_REQ_KEY, string.Empty);
        if (!string.IsNullOrEmpty(req))
        {
            // 2) Siivoa avain heti (one-shot)
            PlayerPrefs.DeleteKey(EDITOR_REQ_KEY);

            // 3) Varmista että kenttä on ladattavissa (Build Settingsissä)
            if (Application.CanStreamedLevelBeLoaded(req))
```

```
            {
                // 4) Ohjaa sekä CurrentLevel että DefaultLevel tähän
                CurrentLevel = req;
                defaultLevel = req;
            }
            else
            {
                Debug.LogWarning($"[LevelLoader] Pyydetty '{req}', mutta sitä ei löydy Build Settingsistä.");
            }
        }
#endif
       // if (string.IsNullOrEmpty(CurrentLevel)) CurrentLevel = defaultLevel;
    }

    public void LoadByIndex(int index)
    {
        StartCoroutine(Co_LoadLocal(index));
    }

    public void Reload() => LoadByIndex(currentIndex);

    private IEnumerator Co_LoadLocal(int index)
    {
        // Unload edellinen, load uusi, set active...
        // Sama pattern kuin NetLevelLoaderissa, mutta ilman RPC:itä
        currentIndex = index;
        yield break;
    }

    public void StartLocalReload(string levelName = null)
    {
        var target = string.IsNullOrWhiteSpace(levelName) ? CurrentLevel ?? defaultLevel : levelName;
        StopAllCoroutines();
    }

    // Kutsu tämä Play Again -napista OFFLINE-tilassa
    public void ReloadOffline(string levelName = null)
    {
        var target = string.IsNullOrWhiteSpace(levelName) ? (CurrentLevel ?? DefaultLevel) : levelName;
        StopAllCoroutines();
        StartCoroutine(Co_ReloadOffline(target));
    }

    private IEnumerator Co_ReloadOffline(string levelName)
    {
        string coreName = CoreSceneName ?? "Core";

        // 0) Lista ennen (debug)
        for (int i = 0; i < SceneManager.sceneCount; i++)
        {
            var s = SceneManager.GetSceneAt(i);
        }
```

```
        // 1) Pura kaikki ei-Core -scenet
        for (int i = 0; i < SceneManager.sceneCount; i++)
        {
            var s = SceneManager.GetSceneAt(i);
            if (!s.isLoaded || s.name == coreName) continue;

            var op = SceneManager.UnloadSceneAsync(s);
            if (op != null) while (!op.isDone) yield return null;

            // koska sceneCount muuttuu, aloita alusta
            i = -1;
        }

        // 2) Varmista Core ladattu + aktiivinen
        var core = SceneManager.GetSceneByName(coreName);
        if (!core.IsValid() || !core.isLoaded)
        {
            var loadCore = SceneManager.LoadSceneAsync(coreName, LoadSceneMode.Additive);
            while (!loadCore.isDone) yield return null;
            core = SceneManager.GetSceneByName(coreName);
        }
        SceneManager.SetActiveScene(core);

        // (siivoa roskat – vapauttaa tuhotun scenen assetteja)
        yield return Resources.UnloadUnusedAssets();
        yield return null;

        // 3) Lataa uusi taso additiivisesti
        var op2 = SceneManager.LoadSceneAsync(levelName, LoadSceneMode.Additive);
        while (!op2.isDone) yield return null;

        var map = SceneManager.GetSceneByName(levelName);
        if (!map.IsValid() || !map.isLoaded)
        {
            Debug.LogError($"[LevelLoader] Failed to load '{levelName}'. Is it in Build Settings?");
            yield break;
        }

        // 4) Aseta map aktiiviseksi yhdeksi frameksi (Start/Awake/OnEnable → placeholderit spawn)
        SceneManager.SetActiveScene(map);
        yield return null;                  // 1 frame
        yield return new WaitForEndOfFrame(); // varmistaa että scene-Startit ehtii

        // 5) Hookit (kuten OfflineBootissa)
        var edgeBaker = UnityEngine.Object.FindFirstObjectByType<EdgeBaker>();
        if (edgeBaker != null) edgeBaker.BakeAllEdges();

        // (valinn.) jos käytössä: miehitys uudelleen sceneen spawneista
        if (LevelGrid.Instance != null) LevelGrid.Instance.RebuildOccupancyFromScene();

        // 6) Core takaisin aktiiviseksi, ilmoita että valmis
```

```
        SceneManager.SetActiveScene(core);
        CurrentLevel = levelName;

        try { RaiseLevelReady(map); } catch { }



        for (int i = 0; i < SceneManager.sceneCount; i++)
        {
            var s = SceneManager.GetSceneAt(i);
            MousePlaneMap.Instance.Rebuild();
        }
    }
}
```

Assets/scripts/LevelCreation/MapContentSpawner.cs

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

public class MapContentSpawner : MonoBehaviour // ⟵ EI NetworkBehaviour
{
    private void Start()
    {
        if (!NetMode.IsServer) return;   // varmistus
        LoSBlockerRegistry.Reset();
        StartCoroutine(SpawnThenBake_ServerOnly());
    }

    private IEnumerator SpawnThenBake_ServerOnly()
    {
        Scene mapScene = gameObject.scene;

        var placeholders = FindObjectsByType<ObjectSpawnPlaceHolder>(FindObjectsSortMode.None);
        int spawned = 0;
        foreach (var sp in placeholders)
        {
            if (sp.gameObject.scene == mapScene)
            {
                var go = sp.CreteObject();   // tämä jo käyttää SpawnRouteria → assetId-prefab spawn, ei sceneId
                if (go) spawned++;
            }
        }

        // Odota, että LevelGrid/PathFinding/EdgeBaker ovat valmiit
        yield return new WaitUntil(() =>
            EdgeBaker.Instance != null &&
            LevelGrid.Instance  != null &&
            PathFinding.Instance != null
        );
        yield return null;

        // Server bake
        EdgeBaker.Instance.BakeAllEdges();
    }
}
```

Assets/scripts/LevelCreation/PostLevelBootstrap.cs

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

/// <summary>
///  Tämä toistaiseksi auttaa vain Solopelissä alussa lataamaan pelaajan yksiköt.
/// </summary>
public class PostLevelBootstrap : MonoBehaviour
{

    private void OnEnable()
    {
        LevelLoader.LevelReady += OnLevelReady;
    }

    private void OnDisable()
    {
        LevelLoader.LevelReady -= OnLevelReady;
    }

    private void OnLevelReady(Scene mapScene)
    {
        if (NetMode.IsRemoteClient) return;
        StartCoroutine(Co_BootstrapAfterLevelReady(mapScene));

    }

    private IEnumerator Co_BootstrapAfterLevelReady(Scene mapScene)
    {
        // Odota 1 frame että Level-skenen Start/OnStartServer ehtivät
        yield return null;

        var spawner = FindFirstObjectByType<SpawnUnitsCoordinator>(FindObjectsInactive.Include);

        if (!spawner)
        {
            Debug.LogError("[Bootstrap] SpawnUnitsCoordinator not found in Level scene.");
            yield break;
        }

        // OFFLINE: (jos haluat tukea SP-tilan bootstrapin tässä)
        if (!NetworkClient.active && !NetworkServer.active)
        {
            // Kutsu suoraan omaa SP-metodiasi, esim:
            spawner.SpawnSinglePlayerUnits();
            LevelGrid.Instance?.RebuildOccupancyFromScene();
            TurnSystem.Instance?.ResetAndBegin();
        }
    }
```

```
}
```

Assets/scripts/LevelCreation/SpawnUnitsCoordinator.cs

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

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

    // ...luokan sisälle:
    [Header("Use placeholders instead of arrays")]
    public bool usePlaceholders = true;

    [Tooltip("Jos true, koordinaattori disabloi/tuhouttaa käytetyt placeholderit serverillä heti spawnin jälkeen.")]
    public bool consumePlaceholdersOnServer = true;

    private bool enemiesSpawned;

    [Header("Co-op squad prefabs")]
    public GameObject unitHostPrefab;
    public GameObject unitClientPrefab;

    [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)
        {
            var myScene = gameObject.scene;
            var oldScene = Instance.gameObject.scene;

            // Jos edellinen on eri scenessä (jäänne purkamatta), tuhoa se ja ota tämä käyttöön
            if (oldScene != myScene)
```

```
            {
                Debug.LogWarning($"[SpawnUnitsCoordinator] Replacing leftover instance from scene '{oldScene.name}' with current '{myScene.name}'.");
                Destroy(Instance.gameObject);
                Instance = this;
                return;
            }

            // Sama scene → tämä on tupla oikeasti: tuhoa tämä
            Debug.LogError($"There's more than one SpawnUnitsCoordinator! {Instance} - {this}");
            Destroy(gameObject);
            return;
        }

        Instance = this;
    }

    void OnDestroy()
    {
        if (Instance == this) Instance = null;
    }

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

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

        var spawnedPlayersUnit = new GameObject[spawnPoints.Length];

        // Hae Level-scene (tämä SpawnUnitsCoordinator on Level-scenessä)
        Scene levelScene = gameObject.scene;

        for (int i = 0; i < spawnPoints.Length; i++)
        {
            // Käytä SpawnRouteria → varmistaa että unitit menevät Level-sceneen
            var playerUnit = SpawnRouter.SpawnNetworkServer(
                prefab: unitPrefab,
                pos: spawnPoints[i],
                rot: Quaternion.identity,
                source: transform,  // Käytä tämän objektin sceneä
                sceneName: levelScene.name,
                parent: null,
```

```
                owner: conn,
                beforeSpawn: (go) =>
                {
                    if (go.TryGetComponent<Unit>(out var u))
                    {
                        if (conn.identity != null) u.OwnerId = conn.identity.netId;

                        // 1) Vision-komponentti (mieluiten valmiiksi prefabissa):
                        if (go.TryGetComponent<UnitVision>(out var uv))
                        {
                            InitUnitVision(go, teamId: (GameModeManager.SelectedMode == GameMode.Versus)
                            ? (NetworkSync.IsOwnerHost(u.OwnerId) ? 0 : 1)
                            : 0);
                        }
                    }
                }
            );

            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 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;
    }

    // Get Spawn positions from placeholders in the scene
    private Vector3[] GetSpawnPositionsFromPlaceholders(UnitSpawnPlaceholder.Side side)
    {
        var scene = gameObject.scene;
        var all = FindObjectsByType<UnitSpawnPlaceholder>(FindObjectsSortMode.None);
        var mine = new List<UnitSpawnPlaceholder>(all.Length);

        foreach (var ph in all)
        {
            if (!ph) continue;
            if (ph.gameObject.scene != scene) continue; // vain tämän Level-skenen placeholderit
            if (ph.side != side) continue;
            mine.Add(ph);
        }

        if (mine.Count == 0) return System.Array.Empty<Vector3>();

        // deterministinen järjestys order-kentän mukaan, sitten nimi/instanceID fallback
        var ordered = mine.OrderBy(p => p.order).ThenBy(p => p.name).ToList();

        var result = new Vector3[ordered.Count];
        for (int i = 0; i < ordered.Count; i++)
            result[i] = ordered[i].GetSpawnWorldPosition();

        // serverillä siivotaan placeholderit jos niin halutaan
        if (consumePlaceholdersOnServer && Mirror.NetworkServer.active)
            foreach (var ph in ordered) ph.Consume();

        return result;
    }

    public Vector3[] GetEnemySpawnPositions()
    {

        if (usePlaceholders)
        {
            var pos = GetSpawnPositionsFromPlaceholders(UnitSpawnPlaceholder.Side.Enemy);
            if (pos.Length > 0) return pos;
            Debug.LogWarning("[SpawnUnitsCoordinator] No enemy placeholders found, falling back to arrays.");
        }

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

```
    public Vector3[] GetSpawnPositionsForPlayer(bool isHost)
    {
        if (usePlaceholders)
        {
            var side = isHost ? UnitSpawnPlaceholder.Side.Host : UnitSpawnPlaceholder.Side.Client;
            var pos = GetSpawnPositionsFromPlaceholders(side);
            if (pos.Length > 0) return pos;
            Debug.LogWarning("[SpawnUnitsCoordinator] No placeholders found, falling back to arrays.");
        }

        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 void SpawnSinglePlayerUnits()
    {
        Scene targetScene = gameObject.scene;

        // PLAYER (Host) – hae paikat placeholdereista (tai fallback taulukkoon)
        var playerSpawns = GetSpawnPositionsForPlayer(true);

        for (int i = 0; i < playerSpawns.Length; i++)
        {
            var unit = SpawnRouter.SpawnLocal(
                prefab: unitHostPrefab,
                pos: playerSpawns[i],
                rot: Quaternion.identity,
                source: transform,
                sceneName: targetScene.name
            );

            InitUnitVision(unit, teamId: 0);
        }

        // ENEMY – samoin placeholdereista (tai fallback)
        var enemySpawns = GetEnemySpawnPositions();

        for (int i = 0; i < enemySpawns.Length; i++)
        {
            var enemy = SpawnRouter.SpawnLocal(
                prefab: GetEnemyPrefab(),
                pos: enemySpawns[i],
                rot: Quaternion.identity,
                source: transform,
                sceneName: targetScene.name
            );

            InitUnitVision(enemy, teamId: 1);
```

```
        }

        SetEnemiesSpawned(true);
    }

    public GameObject[] SpawnEnemies()
    {
        // 1) Hae paikat (placeholderit jos käytössä, muuten fallback-taulukko)
        var enemySpawns = GetEnemySpawnPositions();
        Scene targetScene = gameObject.scene;

        var spawnedEnemies = new GameObject[enemySpawns.Length];

        for (int i = 0; i < enemySpawns.Length; i++)
        {
            if (NetworkServer.active)
            {
                var go = SpawnRouter.SpawnNetworkServer(
                    prefab: GetEnemyPrefab(),
                    pos: enemySpawns[i],
                    rot: Quaternion.identity,
                    source: transform,
                    sceneName: targetScene.name,
                    parent: null,
                    owner: null
                );
                spawnedEnemies[i] = go;

                InitUnitVision(go, teamId: 1);
            }
            else
            {
                var go = SpawnRouter.SpawnLocal(
                    prefab: GetEnemyPrefab(),
                    pos: enemySpawns[i],
                    rot: Quaternion.identity,
                    source: transform,
                    sceneName: targetScene.name
                );
                spawnedEnemies[i] = go;

                InitUnitVision(go, teamId: 1);

            }
        }

        SetEnemiesSpawned(true);
        return spawnedEnemies;
    }

    private void InitUnitVision(GameObject go, int teamId)
    {
```

```
        // Dedi-serverillä ei tarvita paikallista visualisointia
      //  if (NetworkServer.active && !NetworkClient.active) return;

        if (go.TryGetComponent<Unit>(out var u) && go.TryGetComponent<UnitVision>(out var unitVision))
        {
            // Asetetaan oikea tiimi ja alustetaan näkyvyys
            unitVision.InitializeVision(teamId, u.archetype);
        }
        else
        {
            Debug.LogWarning($"[SpawnUnitsCoordinator] {go.name} missing Unit or UnitVision component");
        }
    }



}
```

Assets/scripts/LevelCreation/UnitSpawnPlaceholder.cs

```
using UnityEngine;

#if UNITY_EDITOR
[ExecuteAlways]
#endif
[DisallowMultipleComponent]
public class UnitSpawnPlaceholder : MonoBehaviour
{
    public enum Side { Host, Client, Enemy }

    [Header("Who spawns here?")]
    public Side side = Side.Host;

    [Header("Options")]
    [Tooltip("Snäppää paikan ruudun keskelle, jos LevelGrid on käytössä.")]
    public bool snapToGridCenter = true;

    [Tooltip("Piilota renderöijät play-tilassa (Editorissa näkyy).")]
    public bool hideRendererInPlayMode = true;

    [Tooltip("Tuhotaan palvelimella spawnin jälkeen (muussa tapauksessa disabloidaan).")]
    public bool destroyOnServerAfterUse = true;

    [Tooltip("Vapaa järjestysnumero deterministiseen spawn-järjestykseen (pienin ensin).")]
    public int order = 0;

    public Vector3 GetSpawnWorldPosition()
    {
        var pos = transform.position;

        if (snapToGridCenter && LevelGrid.Instance != null)
        {
            var gp = LevelGrid.Instance.GetGridPosition(pos);
            pos = LevelGrid.Instance.GetWorldPosition(gp); // keskittää ruutuun
        }

        return pos;
    }

    private void OnEnable()
    {
#if UNITY_EDITOR
        if (!Application.isPlaying) return;
#endif
        if (hideRendererInPlayMode) ToggleRenderers(false);
    }

    public void Consume()
    {
        if (destroyOnServerAfterUse)
```

```
        {
            if (Application.isPlaying) Destroy(gameObject);
            else DestroyImmediate(gameObject);
        }
        else
        {
            gameObject.SetActive(false);
        }
    }

    private void ToggleRenderers(bool visible)
    {
        foreach (var r in GetComponentsInChildren<Renderer>(true)) r.enabled = visible;
    }

#if UNITY_EDITOR
    private void OnDrawGizmos()
    {
        // yksinkertainen “keila”: väri puolen mukaan
        Color c = side == Side.Host ? new Color(0f, 0.9f, 1f, 0.9f)
                 : side == Side.Client ? new Color(1f, 0f, 1f, 0.9f)
                 : new Color(1f, 0.2f, 0.2f, 0.9f);
        Gizmos.color = c;
        Gizmos.DrawWireCube(transform.position + Vector3.up * 0.05f, new Vector3(0.6f, 0.1f, 0.6f));
        Gizmos.DrawLine(transform.position, transform.position + Vector3.up * 0.8f);
    }
#endif
}
```

Assets/scripts/LevelCreation/WarmBootGuard.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;
using Mirror;


// Tämä on vain fallback – jos LevelLoader toimii, tämä tekee ei mitään.
public static class WarmBootGuard
{
    /*
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterSceneLoad)]
    static void AfterSceneLoad()
    {
        // Koskee vain offline-starttia. Host/client-tilan latauksen hoitaa NetLevelLoader.
        if (NetworkServer.active || NetworkClient.active) return;

        // jos LevelLoaderia ei jostain syystä löydy tai se ei toimi,
        // lataa varmuuden vuoksi Level 0 additiivisesti.
        var loader = Object.FindFirstObjectByType<LevelLoader>(FindObjectsInactive.Include);
        if (loader == null)
        {
            Debug.LogWarning("[WarmBootGuard] LevelLoader not found → loading 'Level 0' additively as a fallback.");
            SceneManager.LoadSceneAsync("Level 0", LoadSceneMode.Additive);
        }
    }
    */
}
```

Assets/scripts/LevelGridGizmos.cs

```
using UnityEngine;

#if UNITY_EDITOR
using UnityEditor;
using UnityEngine.Rendering;
#endif

[ExecuteInEditMode]
public class LevelGridGizmos : MonoBehaviour
{
    [Header("Visualization Settings")]
    [SerializeField] private bool showGrid = true;
    [SerializeField] private bool showGridNumbers = false;
    [SerializeField] private bool showEdgeWalls = true;
    [SerializeField] private bool showCoverLines = true;

    [Header("Grid Settings (for Edit Mode)")]
    [SerializeField] private int editorWidth = 10;
    [SerializeField] private int editorHeight = 10;
    [SerializeField] private float editorCellSize = 2f;
    [SerializeField] private int editorFloorAmount = 1;

    [Header("Colors")]
    [SerializeField] private Color gridColor = new Color(0.7f, 0.7f, 0.7f, 0.4f);
    [SerializeField] private Color wallColor = new Color(1f, 0f, 0f, 0.8f);
    [SerializeField] private Color lowCoverColor = new Color(1f, 1f, 0f, 0.6f);
    [SerializeField] private Color highCoverColor = new Color(1f, 0.5f, 0f, 0.8f);

    [Header("Line Thickness")]
    [SerializeField] private float gridLineThickness = 1f;
    [SerializeField] private float wallLineThickness = 3f;
    [SerializeField] private float coverLineThickness = 2f;


    public enum DepthMode { XRay, Occluded, Dual }
    [Header("Depth/Overlay")]
    [SerializeField] private DepthMode depthMode = DepthMode.Occluded;
    [SerializeField] private float gridYOffset = 0.01f; // pieni nosto lattian yläpuolelle

    [Header("References")]
    [SerializeField] private LevelGrid levelGrid;

    private const float WALL_HEIGHT_OFFSET = 0.5f;
    private const float COVER_HEIGHT_OFFSET = 0.3f;

    private void OnValidate()
    {
        if (levelGrid == null)
        {
            levelGrid = GetComponent<LevelGrid>();
```

```
        }
    }

    private void OnDrawGizmos()
    {
        if (!showGrid && !showEdgeWalls && !showCoverLines) return;

        if (Application.isPlaying && levelGrid != null)
        {
            DrawPlayModeGizmos();
        }
        else if (!Application.isPlaying)
        {
            DrawEditModeGizmos();
        }
    }

    private static Vector3 Centerize(Vector3 corner, float cell)
    => corner + new Vector3(cell * 0.5f, 0f, cell * 0.5f);

    private void DrawPlayModeGizmos()
    {
        if (levelGrid == null) return;

        int width = 0;
        int height = 0;
        int floorAmount = 0;
        float cellSize = 0f;

        try
        {
            width = levelGrid.GetWidth();
            height = levelGrid.GetHeight();
            floorAmount = levelGrid.GetFloorAmount();
            cellSize = levelGrid.GetCellSize();
        }
        catch
        {
            return;
        }

        if (width <= 0 || height <= 0 || cellSize <= 0) return;

        for (int floor = 0; floor < floorAmount; floor++)
        {
            if (showGrid)
            {
                DrawGridLines(width, height, cellSize, floor);
            }

            if (showGridNumbers)
            {
```

```
                DrawGridNumbers(width, height, floor);
            }

            if (showEdgeWalls || showCoverLines)
            {
                DrawEdgesAndCovers(width, height, floor);
            }
        }
    }

    private void DrawEditModeGizmos()
    {
        if (!showGrid) return;

        for (int floor = 0; floor < editorFloorAmount; floor++)
        {
            DrawGridLines(editorWidth, editorHeight, editorCellSize, floor);

            if (showGridNumbers)
            {
                DrawEditModeGridNumbers(editorWidth, editorHeight, editorCellSize, floor);
            }
        }
    }


    private void DrawGridLines(int width, int height, float cellSize, int floor)
    {
        Gizmos.color = gridColor;
        float floorY = floor * LevelGrid.FLOOR_HEIGHT + gridYOffset; // pieni offset

        float o = 0.5f * cellSize; // siirto että ruudukko alkaa (0,0) vasen/alareunasta

        for (int x = 0; x <= width; x++)
        {
            Vector3 start = new Vector3(x * cellSize - o, floorY, -o);
            Vector3 end = new Vector3(x * cellSize - o, floorY, height * cellSize - o);
            DrawThickLine(start, end, gridLineThickness);
        }

        for (int z = 0; z <= height; z++)
        {
            Vector3 start = new Vector3(-o, floorY, z * cellSize - o);
            Vector3 end = new Vector3(width * cellSize - o, floorY, z * cellSize - o);
            DrawThickLine(start, end, gridLineThickness);
        }
    }

#if UNITY_EDITOR
    // Pieni apu: piirrä viiva depth-asetuksella
    private void DrawWithDepth(System.Action draw)
    {
```

```
        var prev = Handles.zTest;
        switch (depthMode)
        {
            case DepthMode.XRay:
                Handles.zTest = CompareFunction.Always;     // aina näkyvissä
                draw();
                break;
            case DepthMode.Occluded:
                Handles.zTest = CompareFunction.LessEqual;  // kunnioittaa syvyyttä
                draw();
                break;
            case DepthMode.Dual:
                // 1) haalea x-ray taustalle
                Handles.zTest = CompareFunction.Always;
                var c = Gizmos.color; var faint = new Color(c.r, c.g, c.b, c.a * 0.25f);
                Handles.DrawBezier(_s, _e, _s, _e, faint, null, _t);
                // 2) täysi viiva vain näkyvissä osissa
                Handles.zTest = CompareFunction.LessEqual;
                draw();
                break;
        }
        Handles.zTest = prev;
    }

    // Väliaikaiset muuttujat Dual-moodin sisäpiirtoa varten
    private Vector3 _s, _e; private float _t;
#endif
    private void DrawThickLine(Vector3 start, Vector3 end, float thickness)
    {
#if UNITY_EDITOR
        _s = start; _e = end; _t = thickness;
        DrawWithDepth(() =>
        {
            Handles.DrawBezier(start, end, start, end, Gizmos.color, null, thickness);
        });
#else
        Gizmos.DrawLine(start, end);
#endif
    }


    private void DrawGridNumbers(int width, int height, int floor)
    {
    #if UNITY_EDITOR
        float cell = levelGrid.GetCellSize();
        for (int x = 0; x < width; x++)
        for (int z = 0; z < height; z++)
        {
            GridPosition gp = new GridPosition(x, z, floor);
            Vector3 worldPosCorner;
            try { worldPosCorner = levelGrid.GetWorldPosition(gp); }
            catch { continue; }
```

```
            Vector3 worldPos = Centerize(worldPosCorner, cell);
            worldPos.y += 0.1f;

            var style = new GUIStyle { normal = { textColor = Color.white }, fontSize = 10, alignment = TextAnchor.MiddleCenter };
            Handles.Label(worldPos, $"{x},{z}", style);
        }
    #endif
    }

    private void DrawEditModeGridNumbers(int width, int height, float cellSize, int floor)
    {
    #if UNITY_EDITOR
        float floorY = floor * LevelGrid.FLOOR_HEIGHT;
        for (int x = 0; x < width; x++)
        for (int z = 0; z < height; z++)
        {
            Vector3 worldPos = new Vector3((x + 0.5f) * cellSize, floorY, (z + 0.5f) * cellSize);
            worldPos.y += 0.1f;

            var style = new GUIStyle { normal = { textColor = Color.white }, fontSize = 10, alignment = TextAnchor.MiddleCenter };
            Handles.Label(worldPos, $"{x},{z}", style);
        }
    #endif
    }

    private void DrawEdgesAndCovers(int width, int height, int floor)
    {
        var pathfinding = PathFinding.Instance;
        if (pathfinding == null) return;

        float cell = levelGrid.GetCellSize();

        for (int x = 0; x < width; x++)
        for (int z = 0; z < height; z++)
        {
            PathNode node;
            try { node = pathfinding.GetNode(x, z, floor); } catch { continue; }
            if (node == null) continue;

            GridPosition gp = new GridPosition(x, z, floor);
            Vector3 corner;
            try { corner = levelGrid.GetWorldPosition(gp); } catch { continue; }

            // *** TÄRKEÄ: käytä ruudun keskikohtaa piirtämisen lähtöpisteenä
            Vector3 center = Centerize(corner, cell);

            if (showEdgeWalls)  DrawEdgeWalls(node, center, cell);
            if (showCoverLines) DrawCoverLines(node, center, cell);
        }
    }
```

```
    private void DrawEdgeWalls(PathNode node, Vector3 center, float cellSize)
    {
        Gizmos.color = wallColor;
        float halfCell = cellSize * 0.5f;
        float y = center.y + WALL_HEIGHT_OFFSET;

        if (node.HasWall(EdgeMask.N))
        {
            Vector3 start = new Vector3(center.x - halfCell, y, center.z + halfCell);
            Vector3 end = new Vector3(center.x + halfCell, y, center.z + halfCell);
            DrawThickLine(start, end, wallLineThickness);
        }

        if (node.HasWall(EdgeMask.S))
        {
            Vector3 start = new Vector3(center.x - halfCell, y, center.z - halfCell);
            Vector3 end = new Vector3(center.x + halfCell, y, center.z - halfCell);
            DrawThickLine(start, end, wallLineThickness);
        }

        if (node.HasWall(EdgeMask.E))
        {
            Vector3 start = new Vector3(center.x + halfCell, y, center.z - halfCell);
            Vector3 end = new Vector3(center.x + halfCell, y, center.z + halfCell);
            DrawThickLine(start, end, wallLineThickness);
        }

        if (node.HasWall(EdgeMask.W))
        {
            Vector3 start = new Vector3(center.x - halfCell, y, center.z - halfCell);
            Vector3 end = new Vector3(center.x - halfCell, y, center.z + halfCell);
            DrawThickLine(start, end, wallLineThickness);
        }
    }

    private void DrawCoverLines(PathNode node, Vector3 center, float cellSize)
    {
        float halfCell = cellSize * 0.5f;
        float y = center.y + COVER_HEIGHT_OFFSET;

        if (node.HasHighCover(CoverMask.N))
        {
            Gizmos.color = highCoverColor;
            Vector3 start = new Vector3(center.x - halfCell, y, center.z + halfCell);
            Vector3 end = new Vector3(center.x + halfCell, y, center.z + halfCell);
            DrawThickLine(start, end, coverLineThickness);
        }
        else if (node.HasLowCover(CoverMask.N))
        {
            Gizmos.color = lowCoverColor;
            Vector3 start = new Vector3(center.x - halfCell, y, center.z + halfCell);
            Vector3 end = new Vector3(center.x + halfCell, y, center.z + halfCell);
```

```
            DrawThickLine(start, end, coverLineThickness);
        }

        if (node.HasHighCover(CoverMask.S))
        {
            Gizmos.color = highCoverColor;
            Vector3 start = new Vector3(center.x - halfCell, y, center.z - halfCell);
            Vector3 end = new Vector3(center.x + halfCell, y, center.z - halfCell);
            DrawThickLine(start, end, coverLineThickness);
        }
        else if (node.HasLowCover(CoverMask.S))
        {
            Gizmos.color = lowCoverColor;
            Vector3 start = new Vector3(center.x - halfCell, y, center.z - halfCell);
            Vector3 end = new Vector3(center.x + halfCell, y, center.z - halfCell);
            DrawThickLine(start, end, coverLineThickness);
        }

        if (node.HasHighCover(CoverMask.E))
        {
            Gizmos.color = highCoverColor;
            Vector3 start = new Vector3(center.x + halfCell, y, center.z - halfCell);
            Vector3 end = new Vector3(center.x + halfCell, y, center.z + halfCell);
            DrawThickLine(start, end, coverLineThickness);
        }
        else if (node.HasLowCover(CoverMask.E))
        {
            Gizmos.color = lowCoverColor;
            Vector3 start = new Vector3(center.x + halfCell, y, center.z - halfCell);
            Vector3 end = new Vector3(center.x + halfCell, y, center.z + halfCell);
            DrawThickLine(start, end, coverLineThickness);
        }

        if (node.HasHighCover(CoverMask.W))
        {
            Gizmos.color = highCoverColor;
            Vector3 start = new Vector3(center.x - halfCell, y, center.z - halfCell);
            Vector3 end = new Vector3(center.x - halfCell, y, center.z + halfCell);
            DrawThickLine(start, end, coverLineThickness);
        }
        else if (node.HasLowCover(CoverMask.W))
        {
            Gizmos.color = lowCoverColor;
            Vector3 start = new Vector3(center.x - halfCell, y, center.z - halfCell);
            Vector3 end = new Vector3(center.x - halfCell, y, center.z + halfCell);
            DrawThickLine(start, end, coverLineThickness);
        }
    }
}
```

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 TMPro;
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
    [SerializeField] private GameObject connectCodePanel;        // Hiden on start

    [Header("Services")]
    [SerializeField] private Authentication authentication; // <-- UUSI

    [Header("Join Code UI")]
    [SerializeField] private TMP_Text joinCodeText;

    // 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);
        connectCodePanel.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()
    {
        // 0) Varmista että Authentication löytyy (älä luota pelkkään connectCanvas-viitteeseen)
```

```
        if (!authentication)
            authentication = FindFirstObjectByType<Authentication>(FindObjectsInactive.Include);

        if (!authentication)
        {
            Debug.LogError("[GameModeSelectUI] Authentication-componenttia ei löytynyt scenestä.");
            return;
        }

        // 1) Sign-in Unity Servicesiin
        await authentication.SingInPlayerToUnityServerAsync();

        // 2) UI-flown jatko
        FieldCleaner.ClearAll();
        StartCoroutine(ResetGridNextFrame());
        if (gameModeSelectCanvas) gameModeSelectCanvas.SetActive(false);
        if (connectCanvas) 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.RequestReset();
        }
        else
        {
            // Yksinpeli
            GameReset.HardReloadSceneKeepMode();
        }
    }

    public void SetConnectCodePanelVisibility(bool active)
    {
        connectCodePanel.SetActive(active);
    }

    public void SetJoinCodeText(string s)
    {
        if (!joinCodeText)
```

```
        {
            Debug.LogWarning("[GameModeSelectUI] joinCodeText not assigned.");
            return;
        }

        s = (s ?? "").Trim().ToUpperInvariant();
        joinCodeText.text = $"JOIN CODE: {s}";

        // (valinnainen) kopioi koodi leikepöydälle:
        // GUIUtility.systemCopyBuffer = s;

        // (valinnainen) varmista että paneeli on näkyvissä:
        // if (connectCodePanel && !connectCodePanel.activeSelf) connectCodePanel.SetActive(true);
    }
}
```

Assets/scripts/OfflineCalls/OfflineGameSimulator.cs

```
using UnityEngine;

public static class OfflineGameSimulator
{
    public static void SpawnBullet(GameObject bulletPrefab, Vector3 spawnPos, Vector3 targetPos, bool shouldHitUnits)
    {
        SpawnRouter.SpawnLocal(
            bulletPrefab, spawnPos, Quaternion.identity,
            source: null,
            sceneName: LevelLoader.Instance.CurrentLevel,
            parent: null,
            beforeReturn: go =>
            {
                if (go.TryGetComponent<BulletProjectile>(out var gp))
                    gp.Setup(targetPos, shouldHitUnits);
            });
    }

    public static void SpawnGrenade(GameObject grenadePrefab, Vector3 spawnPos, Vector3 targetPos, float maxRangeWU)
    {
        SpawnRouter.SpawnLocal(
            grenadePrefab, spawnPos, Quaternion.identity,
            source: null,
            sceneName: LevelLoader.Instance.CurrentLevel,
            parent: null,
            beforeReturn: go =>
            {
                if (go.TryGetComponent<GrenadeProjectile>(out var gp))
                    gp.ownerTeamId = 0;
                    gp.Setup(targetPos, maxRangeWU);
            });
    }

    public static void SpawnRagdoll(GameObject prefab, Vector3 pos, Quaternion rot, uint sourceUnitNetId, Transform originalRootBone, Vector3 lastHitPosition, int overkill)
    {
        SpawnRouter.SpawnLocal(
            prefab, pos, rot,
            source: originalRootBone,
            sceneName: null,
            parent: null,
            beforeReturn: go =>
            {
                if (go.TryGetComponent<UnitRagdoll>(out var unitRagdoll))
                {
                    unitRagdoll.SetOverkill(overkill);
                    unitRagdoll.SetLastHitPosition(lastHitPosition);
                    unitRagdoll.Setup(originalRootBone);
                }
            });
    }
```

```
}
```

Assets/scripts/Oneline/Connect/ClientPreJoinCleaner.cs

```
// ClientPreJoinCleaner.cs
using System.Collections;
using System.Linq;
using Mirror;
using UnityEngine;
using UnityEngine.SceneManagement;

public static class ClientPreJoinCleaner
{
    public static IEnumerator PrepareForOnlineJoin()
    {
        // 1) Pura kaikki ei-Core -scenet pois
        string core = LevelLoader.Instance ? LevelLoader.Instance.CoreSceneName : "Core";
        for (int i = SceneManager.sceneCount - 1; i >= 0; i--)
        {
            var s = SceneManager.GetSceneAt(i);
            if (s.isLoaded && s.name != core)
            {
                var op = SceneManager.UnloadSceneAsync(s);
                if (op != null) yield return op;
            }
        }

        // 2) Siivoa mahdolliset offline-jäänteet Coresta,
        //    jotka pitäisi tulla serveriltä netin kautta
        DestroyServerProvidedLeftoversInCore(core);

        // (valinnainen)
         yield return Resources.UnloadUnusedAssets();
    }

    static void DestroyServerProvidedLeftoversInCore(string core)
    {
        var coreScene = SceneManager.GetSceneByName(core);
        if (!coreScene.IsValid()) return;

        // Hae kaikki juuret Core-scenen alta ja siivoa tunnetut tyypit
        foreach (var root in coreScene.GetRootGameObjects())
        {
            // Unitit (mukaan lukien ystävä/vihollis-tagit)
            foreach (var u in root.GetComponentsInChildren<Unit>(true)) Object.Destroy(u.gameObject);
            foreach (var f in root.GetComponentsInChildren<FriendlyUnit>(true)) Object.Destroy(f.gameObject);
            foreach (var e in root.GetComponentsInChildren<EnemyUnit>(true)) Object.Destroy(e.gameObject);

            // Tuhoutuvat objektit
            foreach (var d in root.GetComponentsInChildren<DestructibleObject>(true)) Object.Destroy(d.gameObject);

            // Ragdollit & binderit
            foreach (var rb in root.GetComponentsInChildren<RagdollPoseBinder>(true)) Object.Destroy(rb.gameObject);
            foreach (var rd in root.GetComponentsInChildren<UnitRagdoll>(true)) Object.Destroy(rd.gameObject);
```

```
            // (Tarvittaessa: ohjukset/granaatit, placeholderit yms.)
            // foreach (var b in root.GetComponentsInChildren<BulletProjectile>(true)) Object.Destroy(b.gameObject);
            // foreach (var g in root.GetComponentsInChildren<GrenadeProjectile>(true)) Object.Destroy(g.gameObject);
            // foreach (var ph in root.GetComponentsInChildren<ObjectSpawnPlaceHolder>(true)) Object.Destroy(ph.gameObject);
        }

        // Nollaa ruudukon miehitykset varmuudeksi
        LevelGrid.Instance?.ClearAllOccupancy();
    }
}
```

Assets/scripts/Oneline/Connect/Connect.cs

```
using UnityEngine;
using TMPro;
using Mirror;
using Utp;
using UnityEngine.SceneManagement;
using System.Collections;
using UnityEngine.UI;

/// <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;
    [SerializeField] private GameModeSelectUI gameModeSelectUI;

    [SerializeField] private GameObject joinInputPanel;   // JoinInputPanel (inactive alussa)
    [SerializeField] private TMP_InputField joinCodeField;
    [SerializeField] private Button joinButton;

    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.");

        if (joinInputPanel) joinInputPanel.SetActive(false);
        if (joinButton) joinButton.onClick.AddListener(JoinWithFieldValue);
        if (joinCodeField) joinCodeField.onSubmit.AddListener(_ => JoinWithFieldValue());

    }

    public void HostLAN()
    {
        if (!gameNetworkManager)
        {
            gameNetworkManager = NetworkManager.singleton as GameNetworkManager
                              ?? FindFirstObjectByType<GameNetworkManager>();
            if (!gameNetworkManager) { Debug.LogError("[Connect] GameNetworkManager not found."); return; }
        }

        gameNetworkManager.StartStandardHost();

    }
```

```
    public void ClientLAN()
    {
        string ip = (ipField != null && !string.IsNullOrWhiteSpace(ipField.text))
                      ? ipField.text.Trim()
                      : "localhost"; // tai 127.0.0.1

        Debug.Log($"[Connect] Joining server at {ip}");

        gameNetworkManager.networkAddress = ip;
        // 1) Puhdista clientin oma kenttä ja offline-jäänteet
        StartCoroutine(CleanThenJoin());

    }

    private IEnumerator CleanThenJoin()
    {
        // 1) Puhdista clientin oma kenttä ja offline-jäänteet
        yield return ClientPreJoinCleaner.PrepareForOnlineJoin();
        gameNetworkManager.JoinStandardServer();  // useRelay=false ja StartClient()

    }

    public void Host()
    {
        if (!gameNetworkManager)
        {
            gameNetworkManager = NetworkManager.singleton as GameNetworkManager
                           ?? FindFirstObjectByType<GameNetworkManager>();
            if (!gameNetworkManager) { Debug.LogError("[Connect] GameNetworkManager not found."); return; }
        }

        StartCoroutine(StartRelayHostThenLoadLevel());
    }

    private RelayJoinCodeUI GetJoinUI()
    {
        // Etsi ensin singletonista, jos puuttuu, etsi hierarkiasta (myös inaktiivisista)
        return RelayJoinCodeUI.Instance
            ?? FindFirstObjectByType<RelayJoinCodeUI>(FindObjectsInactive.Include);
    }

    private IEnumerator StartRelayHostThenLoadLevel()
    {
        if (NetworkServer.active) yield break;

        gameNetworkManager.StartRelayHost(2, null);

        // 1) Odota että oikea join-koodi valmistuu
        yield return new WaitUntil(() => !string.IsNullOrEmpty(gameNetworkManager.relayJoinCode));

        // 2) Näytä koodi turvallisesti (ei NRE:tä vaikka Instance olisi vielä null)
        var ui = GetJoinUI();
```

```
        if (ui != null) ui.ShowCode(gameNetworkManager.relayJoinCode);
        else Debug.LogError("[Connect] RelayJoinCodeUI puuttuu Corescenestä – ei voida näyttää koodia.");

        // 3) Odota että serveri on varmasti aktiivinen
        yield return new WaitUntil(() => NetworkServer.active);

        // 4) Varmista että level on ladattu additiivisesti
        yield return EnsureLevelLoadedAfterServerUp();

        // 5) Pidä koodi näkyvissä kunnes 2. pelaaja on mukana (host + 1 client)
        var minConn = gameNetworkManager ? gameNetworkManager.HideJoinCodeAfterConnections : 2;
        yield return new WaitUntil(() => NetworkServer.connections != null &&
                                         NetworkServer.connections.Count >= minConn);

        HideJoinPanel();
        if (ui != null) ui.Hide();
    }


    private IEnumerator EnsureLevelLoadedAfterServerUp()
    {
        // Odota että serveri on oikeasti ylhäällä
        yield return new WaitUntil(() => NetworkServer.active);
        yield return null; // 1 frame väliin, että Core-komponentit ehtivät herätä

        // Jos taso ei ole vielä valmis → kysy lataus NetLevelLoaderilta
        if (!LevelLoader.IsServerLevelReady)
        {
            // 1) Jos LevelLoader kertoo nykyisen tai oletustason, käytä sitä
            string desired = LevelLoader.Instance
                ? (LevelLoader.Instance.CurrentLevel ?? LevelLoader.Instance.DefaultLevel)
                : null;

            // 2) Muuten pyydä NetLevelLoaderilta sen oletustaso
            if (string.IsNullOrEmpty(desired) && NetLevelLoader.Instance)
                desired = NetLevelLoader.Instance.ResolveDefaultLevelName();

            if (!string.IsNullOrEmpty(desired) && NetLevelLoader.Instance)
            {
                NetLevelLoader.Instance.ServerLoadLevel(desired);
            }
            else
            {
                Debug.LogError("[Connects] Ei pystytty ratkaisemaan ladattavaa leveliä: puuttuuko LevelLoader.DefaultLevel tai NetLevelLoader?");
            }
        }
    }

    public void Client()
    {

        if (!gameNetworkManager)
```

```
        {
            gameNetworkManager = NetworkManager.singleton as GameNetworkManager
                                 ?? FindFirstObjectByType<GameNetworkManager>();
            if (!gameNetworkManager)
            {
                Debug.LogError("[Connect] GameNetworkManager not found.");
                return;
            }
        }

        ShowJoinPanel();

    }

    private void JoinWithFieldValue()
    {
        if (!gameNetworkManager)
        {
            Debug.LogError("[Connect] GameNetworkManager not set.");
            return;
        }

        string code = (joinCodeField ? joinCodeField.text : "").Trim().ToUpperInvariant();
        if (string.IsNullOrEmpty(code) || code.Length != 6)
        {
            Debug.LogWarning("[Connect] Join code missing/invalid.");
            return;
        }

        // Käynnistä join-prosessi yhdestä paikasta (coroutinesta)
        StartCoroutine(Co_CleanThenJoin(code));
    }

    private IEnumerator Co_CleanThenJoin(string code)
    {
        // Piilota UI heti kun liitytään (ettei jää päälle kentän latautuessa)
        HideJoinPanel();
        if (joinButton) joinButton.interactable = false;

        // Puhdista ennen liittymistä
        yield return ClientPreJoinCleaner.PrepareForOnlineJoin();

        // Aseta koodi ja liity (vain KERRAN)
        gameNetworkManager.relayJoinCode = code;
        gameNetworkManager.JoinRelayServer();
    }

    // Cancel tai Back
    public void HideJoinPanel()
    {
        if (joinInputPanel) joinInputPanel.SetActive(false);
        if (joinCodeField) { joinCodeField.text = ""; joinCodeField.DeactivateInputField(); }
```

```
    }

    private void ShowJoinPanel()
    {
        if (joinInputPanel) joinInputPanel.SetActive(true);
        if (joinCodeField)
        {
            // (valinnainen) esitäyttö leikepöydästä, jos näyttää koodilta
            var clip = GUIUtility.systemCopyBuffer?.Trim().ToUpperInvariant();
            if (!string.IsNullOrEmpty(clip) && clip.Length == 6)
                joinCodeField.text = clip;

            joinCodeField.ActivateInputField();
            joinCodeField.caretPosition = joinCodeField.text.Length;
        }
    }

    /// <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()
    {
        StartCoroutine(StartRelayHostThenChangeScene());
    }

    private IEnumerator StartRelayHostThenChangeScene()
    {

        if (NetworkServer.active) yield break;

        gameNetworkManager.StartRelayHost(2, null);

        // 1) Odota kunnes OIKEA relay-join-koodi on valmis
        yield return new WaitUntil(() => !string.IsNullOrEmpty(gameNetworkManager.relayJoinCode));
        RelayJoinCodeUI.Instance.ShowCode(gameNetworkManager.relayJoinCode);

        // 2) Odota kunnes serveri on aktiivinen
        yield return new WaitUntil(() => NetworkServer.active);

        // 3) Pidä koodi näkyvissä kunnes 2. pelaaja on mukana (host + 1 client)
        yield return new WaitUntil(() =>
            NetworkServer.connections != null && NetworkServer.connections.Count >= 2);
```

```
            RelayJoinCodeUI.Instance.Hide();
    }
}
```

Assets/scripts/Oneline/Connect/GameNetworkManager.cs

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

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

  private readonly List<NetworkConnectionToClient> _pendingConns = new();

  [SerializeField] private int hideJoinCodeAfterConnections = 2; // Host + 1 client

  public int HideJoinCodeAfterConnections => Mathf.Max(1, hideJoinCodeAfterConnections);

  private UtpTransport utpTransport;

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


  public override void Awake()
  {
   if (Instance != null && Instance != this)
   {
    Debug.LogError("There's more than one GameNetworkManager! " + transform + " - " + Instance);
    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();
 LevelLoader.LevelReady += OnLevelReady_Server;

 SpawnUnitsCoordinator.Instance.SetEnemiesSpawned(false);
}

public override void OnStopServer()
{
 LevelLoader.LevelReady -= OnLevelReady_Server;
 base.OnStopServer();
}
/*
void OnEnable()
{
 LevelLoader.LevelReady += OnLevelReady_Server;
}
*/

void OnDisable() { LevelLoader.LevelReady -= OnLevelReady_Server; }

[ServerCallback]
public override void OnDestroy()
{
 LevelLoader.LevelReady -= OnLevelReady_Server;
}

[Server]
private void OnLevelReady_Server(Scene mapScene)
{
 if (!NetworkServer.active) return;

 // 1) Ensilataus: pending-jonon finalisointi
 foreach (var c in _pendingConns)
  if (c != null) ServerFinalizeAddPlayer(c);
 _pendingConns.Clear();
```

```
foreach (var kv in NetworkServer.connections)
{
 var conn = kv.Value;
 if (conn == null) continue;

 // Jos identity on null, luo se uudelleen
 if (conn.identity == null)
 {
  if (playerPrefab != null)
  {
   base.OnServerAddPlayer(conn);
  }
  else
  {
   Debug.LogError("[GameNetworkManager] PlayerPrefab is null!");
   continue;
  }
 }

 // Tarkista onko uniteja
 uint ownerId = conn.identity != null ? conn.identity.netId : 0u;
 bool hasUnits = ownerId != 0 && HasOwnedUnit(ownerId);

 if (!hasUnits)
 {
  bool isHost = conn == NetworkServer.localConnection;

  var units = SpawnUnitsCoordinator.Instance?.SpawnPlayersForNetwork(conn, isHost);
  if (units == null)
  {
   Debug.LogWarning($"[GameNetworkManager] Failed to spawn units for conn {conn.connectionId}");
  }

 }
}

// 3) Viholliset jos tarvitaan
if (GameModeManager.SelectedMode == GameMode.CoOp)
{
 if (!SpawnUnitsCoordinator.Instance.AreEnemiesSpawned())
 {
  ServerSpawnEnemies();
 }
}

// 4) Käynnistä uusi matsi
LevelGrid.Instance?.RebuildOccupancyFromScene();
EdgeBaker.Instance?.BakeAllEdges();
MousePlaneMap.Instance.Rebuild();
NetTurnManager.Instance?.ServerResetAndBegin();
if (NetworkVisionRpcHub.Instance != null)
```

```
        NetworkVisionRpcHub.Instance.RpcResetLocalVisionAndRebuild();
}


[Server]
private bool HasOwnedUnit(uint ownerNetId)
{
 var units = FindObjectsByType<Unit>(FindObjectsSortMode.None);
 for (int i = 0; i < units.Length; i++)
  if (units[i] && units[i].OwnerId == ownerNetId)
   return true;
 return false;
}

/// <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($"Relay join code: {joinCode}");
  Debug.Log($"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();
}

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

 Debug.Log($"[NM] OnClientSceneChanged - ready: {NetworkClient.ready}");

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

 // Pyydä pelaaja jos ei vielä ole
 if (NetworkClient.connection != null && NetworkClient.connection.identity == null)
 {
  NetworkClient.AddPlayer();
 }
}

public override void OnClientConnect()
{
 base.OnClientConnect();

 // Varmista että client on ready
 if (!NetworkClient.ready)
 {
  NetworkClient.Ready();
 }

 // Pyydä pelaaja heti
 if (NetworkClient.connection != null && NetworkClient.connection.identity == null)
 {
  NetworkClient.AddPlayer();
 }
}

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

public override void OnClientDisconnect()
{
 base.OnClientDisconnect();
}
```

```
/// <summary>
/// Tämä metodi spawnaa jokaiselle clientille oman Unitin ja tekee siitä heidän ohjattavan yksikkönsä.
/// </summary>
public override void OnServerAddPlayer(NetworkConnectionToClient conn)
{
 if (!LevelLoader.IsServerLevelReady)
 {
  _pendingConns.Add(conn);
  return;
 }
 ServerFinalizeAddPlayer(conn);
}


[Server]
private void ServerFinalizeAddPlayer(NetworkConnectionToClient conn)
{
 if (conn.identity == null)
 {
  if (playerPrefab == null)
  {
   Debug.LogError("[NM] Player Prefab puuttuu!");
   return;
  }
  base.OnServerAddPlayer(conn);
 }

 bool isHost = conn.connectionId == 0;

 var spawner = SpawnUnitsCoordinator.Instance;
 if (spawner == null)
 {
  Debug.LogError("[NM] SpawnUnitsCoordinator.Instance puuttuu!");
  return;
 }

 spawner.SpawnPlayersForNetwork(conn, isHost);


 var turnMgr = NetTurnManager.Instance;
 if (turnMgr != null)
  turnMgr.ServerUpdateRequiredCount(NetworkServer.connections.Count);

 if (NetTurnManager.Instance && NetTurnManager.Instance.phase == TurnPhase.Players)
 {
  var pc = conn.identity ? conn.identity.GetComponent<PlayerController>() : null;
  if (pc != null) pc.ServerSetHasEnded(false);
 }

 if (CoopTurnCoordinator.Instance && NetTurnManager.Instance)
 {
  CoopTurnCoordinator.Instance.RpcTurnPhaseChanged(
```

```
    NetTurnManager.Instance.phase,
    NetTurnManager.Instance.turnNumber,
    true
   );
  }

  if (GameModeManager.SelectedMode == GameMode.Versus)
  {
   var pc = conn.identity ? conn.identity.GetComponent<PlayerController>() : null;
   if (pc != null && PvPTurnCoordinator.Instance != null)
    PvPTurnCoordinator.Instance.ServerRegisterPlayer(pc);
  }
 }

 [Server]
 public void ServerSpawnEnemies()
 {
  var enemies = SpawnUnitsCoordinator.Instance.SpawnEnemies();

  foreach (var enemy in enemies)
  {
   if (enemy == null) continue;

   NetworkServer.Spawn(enemy); // olemassa oleva rivi

   // UUSI: pakota oikea alkulayout
   var vis = enemy.GetComponentInChildren<WeaponVisibilitySync>();
   if (vis) vis.ServerForceSet(rifleRight: true, rifleLeft: false, meleeLeft: false, grenade: false);
  }
 }

 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/Connect/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();
        // 4b) Varmista että edge/cover-data on synkassa occupancy/geometryn kanssa
        EdgeBaker.Instance.BakeAllEdges();
        // Piilota kaikki käyttämättömät gridit joissa ei ole mouseplane
        MousePlaneMap.Instance.Rebuild();
```

```
        // 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/GamePlay/CoopTurnCoordinator.cs

```
using System.Collections;
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()
    {
        UnitUIBroadcaster.Instance.BroadcastUnitWorldUIVisibility(true);

        // Vihollisvuoro alkaa domainissa
        TurnSystem.Instance.BeginEnemyTurn(incrementTurnId:false);
        RpcTurnPhaseChanged(TurnPhase.Enemy, NetTurnManager.Instance.turnNumber, false);

        // Aja AI
        yield return RunEnemyAI();

        // Siirtymä pelaajille
        NetTurnManager.Instance.turnNumber++;
        NetTurnManager.Instance.ResetTurnState();

        TurnSystem.Instance.BeginPlayersTurn(incrementTurnId:true); // laukoo eventit coresta
        RpcTurnPhaseChanged(TurnPhase.Players, NetTurnManager.Instance.turnNumber, true);

        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/GamePlay/EndgameAnnouncer.cs

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

public class EndgameAnnouncer : NetworkBehaviour
{
    public override void OnStartServer()
    {
        Unit.OnAnyUnitDead += OnAnyUnitDead_Server;
    }
    public override void OnStopServer()
    {
        Unit.OnAnyUnitDead -= OnAnyUnitDead_Server;
    }

    [ServerCallback]
    private void OnAnyUnitDead_Server(object sender, EventArgs e)
    {
        var um = UnitManager.Instance;
        if (um == null) return;

        int friendly = um.GetFriendlyUnitList().Count; // hostin puoli
        int enemy = um.GetEnemyUnitList().Count;    // ei-hostin puoli

        bool end = (enemy <= 0) || (friendly <= 0);
        if (!end) return;

        bool hostWon = enemy <= 0;

        // Lähetä kullekin clientille oma viesti
        foreach (var kv in NetworkServer.connections)
        {
            var conn = kv.Value;
            if (conn?.identity == null) continue;

            var pc = conn.identity.GetComponent<PlayerController>();
            if (pc == null) continue;

            bool isHostConn = conn.connectionId == 0;
            bool youWon = hostWon ? isHostConn : !isHostConn;

            pc.TargetShowEnd(conn, youWon);
        }
    }
}
```

Assets/scripts/Oneline/GamePlay/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
        }
    }

    /// <summary>
/// Serverillä: nollaa vuorot ja aloittaa Players-vaiheen. Kutsutaan aina kun leveli latautuu (myös reloadissa).
/// </summary>
/// <param name="resetTurnNumber">Jos true, turnNumber asetetaan 1:een. Jos false, säilytetään nykyinen (tai voit itse inkrementoida muualla).</param>
    [Server]
    public void ServerResetAndBegin(bool resetTurnNumber = true)
    {
        // Co-op: laske tällä hetkellä aktiiviset pelaajat ja päivitä requiredCount
        if (GameModeManager.SelectedMode == GameMode.CoOp)
        {
            int playersNow = 0;
            foreach (var kv in NetworkServer.connections)
                if (kv.Value != null && kv.Value.identity != null) playersNow++;

            ServerUpdateRequiredCount(playersNow);
        }

        if (resetTurnNumber)
            turnNumber = 1;

        if (GameModeManager.SelectedMode == GameMode.Versus && PvPTurnCoordinator.Instance)
        {
            PvPTurnCoordinator.Instance.ServerGiveFirstTurnToHost();
        }

        ResetTurnState();
    }
}
```

Assets/scripts/Oneline/GamePlay/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/GamePlay/PvPTurnCoordinator.cs

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

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

    [SyncVar] public 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);

        if (TurnSystem.Instance != null)
        TurnSystem.Instance.NextTurn();
    }

    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;
        }
    }

    [Server]
    public void ServerGiveFirstTurnToHost()
    {
        // heti kun ensimmäinen vuoro on annettu hostille:
        if (TurnSystem.Instance != null)
        {
            TurnSystem.Instance.ResetTurnId();                // siisti lähtö (0)
            TurnSystem.Instance.ResetAndBegin(false, true);   // älä koske turnNumberiin; aloita Players-vaiheesta
        }

        var players = GetAllPlayers().ToList();
        var host = players.FirstOrDefault(p => p.connectionToClient != null && p.connectionToClient.connectionId == 0);
```

```
        if (host == null) return;

        currentOwnerNetId = host.netId;

        // Host saa toimia, vastustaja lukkoon
        foreach (var p in players)
            p.ServerSetHasEnded(p != host);

        RpcTurnChanged(GetTurnNumber(), currentOwnerNetId);
    }

}
```

Assets/scripts/Oneline/LevelAndScenes/NetLevelLoader.cs

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

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

    [SyncVar(hook = nameof(OnLevelChanged))]
    private string _currentLevel;

    [SerializeField] private string _fallbackDefaultLevelName; // valinnainen: aseta Inspectorissa jos haluat


    private int _reloadTick = 0;
    private readonly HashSet<int> _clientReadyAcks = new HashSet<int>();

    private static bool _clientIsLoading;
    private static string _clientPreparedLevel;

    [Header("Catalog")]
    [SerializeField] private LevelCatalog catalog;
    [SerializeField] private int currentIndex = -1;  // tällä hetkellä ladattu kartta (katalogin indeksi)

    public int CurrentIndex => currentIndex;

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

    public override void OnStartServer()
    {
        base.OnStartServer();

        int idx = ResolveDefaultIndex();
        var sceneName = catalog.Get(idx).sceneName;
        _currentLevel = sceneName;
        currentIndex = idx;
        StartCo(Co_LoadLevel(idx));
    }
```

```
    public override void OnStartClient()
    {
        base.OnStartClient();
    }

    void OnLevelChanged(string oldValue, string newValue)
    {
        if (isServer) return;
        if (string.IsNullOrEmpty(newValue)) return;
        if (newValue.StartsWith("__RELOAD_TICK__")) return;

        if (_clientPreparedLevel == newValue)
        {
            _clientPreparedLevel = null;
            return;
        }
        StartCoroutine(Co_LoadLevel_Client(newValue));
    }

    [Server]
    public void ServerLoadLevel(string levelName)
    {
        if (string.IsNullOrEmpty(levelName))
        {
            Debug.LogError("[NetLevelLoader] ServerLoadLevel sai tyhjän scenenimen.");
            return;
        }

        StopAllCoroutines();
        StartCo(Co_ReloadLevel_All(levelName));
    }

    [Server]
    private IEnumerator Co_ReloadLevel_All(string levelName)
    {
        string coreName = LevelLoader.Instance?.CoreSceneName ?? "Core";

        _reloadTick++;
        _clientReadyAcks.Clear();

        _currentLevel = $"__RELOAD_TICK__{_reloadTick}";
        RpcClientPrepareReload(coreName, levelName, _reloadTick);

        int expectedClients = ExpectedClientCount();
        float timeout = 15f;
        float elapsed = 0f;

        while (_clientReadyAcks.Count < expectedClients && elapsed < timeout)
        {
            yield return null;
            elapsed += Time.deltaTime;
        }
```

```
        if (elapsed >= timeout)
        {
            Debug.LogWarning($"[NetLevelLoader] (SERVER) Timeout! Got {_clientReadyAcks.Count}/{expectedClients} acks");
        }
        else
        {
            Debug.Log($"[NetLevelLoader] (SERVER) All {_clientReadyAcks.Count} clients ready!");
        }

        int idx = IndexOfScene(levelName);
        if (idx < 0) idx = ResolveDefaultIndex();

        _currentLevel = levelName;
        yield return StartCoroutine(Co_LoadLevel(idx));
    }


    private IEnumerator Co_LoadLevel(int index)
    {
        // 0) Siivous + unload edellinen (jo teillä koodissa)
        UnitManager.Instance?.ClearAllUnitLists();

        var current = CurrentSceneName;
        if (!string.IsNullOrEmpty(current)) {
            var s = SceneManager.GetSceneByName(current);
            if (s.isLoaded) {
                var opUnload = SceneManager.UnloadSceneAsync(s);
                while (!opUnload.isDone) yield return null;
            }
        }

        // 1) Lataa uusi additiivisesti (jo teillä)
        var entry = catalog.Get(index);
        var sceneName = entry.sceneName;
        _currentLevel = sceneName;

        var opLoad = SceneManager.LoadSceneAsync(sceneName, LoadSceneMode.Additive);
        while (!opLoad.isDone) yield return null;

        // 2) Aseta map aktiiviseksi ja anna 1–2 framea heräämiseen
        Scene newScene = SceneManager.GetSceneByName(sceneName);
        SceneManager.SetActiveScene(newScene);
        yield return null;

        NetworkServer.SpawnObjects();       // pakollinen additiivisen scenen scene-objekteille
        yield return null;
        EdgeBaker.Instance?.BakeAllEdges();
        MousePlaneMap.Instance.Rebuild();

        // 3) Päivitä indeksi vasta onnistumisen jälkeen (jo teillä)
        currentIndex = index;
```

```
    // 4) Palauta Core aktiiviseksi jos haluat samaan tapaan kuin toisessa polussa
    var coreName = LevelLoader.Instance?.CoreSceneName ?? "Core";
    var core = SceneManager.GetSceneByName(coreName);
    if (core.IsValid() && core.isLoaded)
        SceneManager.SetActiveScene(core);

    // 5) Ilmoita että servupuoli on valmis → käynnistää OnLevelReady_Server-ketjun
    LevelLoader.SetServerLevelReady(true);
    LevelLoader.RaiseLevelReady(newScene);   // 🔔 tämä käynnistää GameNetworkManagerin spawnit

    // 6) (valinn.) UI-siivo RPC: kuten teillä jo on
    RpcOnLevelLoaded(sceneName, currentIndex);
}

[ClientRpc]
private void RpcClientPrepareReload(string coreName, string levelName, int tick)
{
    if (isServer) return;

    StartCoroutine(Co_ClientPrepareAndAck(coreName, levelName, tick));
}

[Client]
private IEnumerator Co_ClientPrepareAndAck(string coreName, string levelName, int tick)
{

    DebrisUtil.DestroyAllDebrisExceptCore(coreName);

    yield return Co_LoadLevel_Client_Internal(levelName);

    _clientPreparedLevel = levelName;

    CmdAckSceneReady(tick);
}

[Command(requiresAuthority = false)]
void CmdAckSceneReady(int tick, NetworkConnectionToClient sender = null)
{
    if (tick != _reloadTick || sender == null) return;
    _clientReadyAcks.Add(sender.connectionId);
}

[Server]
int ExpectedClientCount()
{
    int c = 0;
    foreach (var kv in NetworkServer.connections)
    {
        if (kv.Value != null && kv.Value.isAuthenticated)
        {
            if (kv.Value == NetworkServer.localConnection)
```

```
                continue;
            c++;
        }
    }
    return c;
}

[Client]
private IEnumerator Co_LoadLevel_Client(string levelName)
{
    if (_clientIsLoading)
    {
        yield break;
    }

    _clientIsLoading = true;

    try
    {
        yield return Co_LoadLevel_Client_Internal(levelName);
    }
    finally
    {
        _clientIsLoading = false;
    }
}

[Client]
private IEnumerator Co_LoadLevel_Client_Internal(string levelName)
{
    string coreName = LevelLoader.Instance?.CoreSceneName ?? "Core";

    var core = SceneManager.GetSceneByName(coreName);
    if (!core.IsValid() || !core.isLoaded)
    {
        var loadCore = SceneManager.LoadSceneAsync(coreName, LoadSceneMode.Additive);
        while (!loadCore.isDone) yield return null;
        core = SceneManager.GetSceneByName(coreName);
    }
    SceneManager.SetActiveScene(core);

    for (int i = 0; i < SceneManager.sceneCount; i++)
    {
        var s = SceneManager.GetSceneAt(i);
        if (!s.isLoaded || s.name == coreName) continue;

        yield return SceneManager.UnloadSceneAsync(s);
        i = -1;
    }

    var op2 = SceneManager.LoadSceneAsync(levelName, LoadSceneMode.Additive);
    while (!op2.isDone) yield return null;
```

```
        var map = SceneManager.GetSceneByName(levelName);
        if (!map.IsValid() || !map.isLoaded)
        {
            Debug.LogError($"[NetLevelLoader] (CLIENT) Failed to load '{levelName}'. Is it in Build Settings?");
            yield break;
        }

        SceneManager.SetActiveScene(map);
        yield return null;
        SceneManager.SetActiveScene(core);
        LevelLoader.RaiseLevelReady(map);
    }

    private void StartCo(IEnumerator r)
    {
        if (isActiveAndEnabled) StartCoroutine(r);
    }

    [Server]
    public void ServerReloadCurrentLevel()
    {
        if (currentIndex < 0) currentIndex = ResolveDefaultIndex();

        var entry = catalog.Get(currentIndex);
        if (entry == null)
        {
            Debug.LogError($"[NetLevelLoader] Cannot reload - invalid index {currentIndex}");
            return;
        }

        string sceneName = entry.sceneName;

        StopAllCoroutines();
        StartCo(Co_ReloadLevel_All(sceneName));
    }

    public string CurrentSceneName =>
        (catalog != null && catalog.Count > 0) ? catalog.Get(currentIndex)?.sceneName : null;



    // Julkinen entry point: lataa nimen perusteella
    [Server]
    public void ServerLoadLevelByName(string sceneName) {
        if (catalog == null) { Debug.LogError("[NetLevelLoader] Catalog puuttuu"); return; }
        int i = catalog.IndexOfScene(sceneName);
        if (i < 0) { Debug.LogError($"[NetLevelLoader] Scene '{sceneName}' ei löydy catalogista"); return; }
        ServerLoadLevelByIndex(i);
    }

    // Julkinen entry point: lataa indeksillä
```

```
    [Server]
    public void ServerLoadLevelByIndex(int index) {
        if (catalog == null || catalog.Count == 0) { Debug.LogError("[NetLevelLoader] Catalog tyhjä"); return; }
        if (index < 0 || index >= catalog.Count) { Debug.LogError("[NetLevelLoader] Index out of range"); return; }

        StartCoroutine(Co_LoadLevel(index));
    }

    [Server]
    public void ServerLoadNextLevelLoop() {
        if (catalog == null || catalog.Count == 0) return;
        int next = (currentIndex + 1) % catalog.Count;
        ServerLoadLevelByIndex(next);
    }

    [ClientRpc]
    void RpcOnLevelLoaded(string sceneName, int index)
    {
        // Client-päässä: UI/HUD siivous, WinPanel piiloon jne.
        var win = FindFirstObjectByType<WinBattle>(FindObjectsInactive.Include);
        if (win) win.HideEndPanel();
    }

    public string ResolveDefaultLevelName()
    {
        // 1) Jos _currentLevel on jo asetettu (esim. edellisestä pelistä / valikosta)
        if (!string.IsNullOrEmpty(_currentLevel)) return _currentLevel;

        // 2) Jos LevelLoaderissa on määritelty oletus
        if (LevelLoader.Instance && !string.IsNullOrEmpty(LevelLoader.Instance.DefaultLevel))
            return LevelLoader.Instance.DefaultLevel;

        // 3) Lopuksi oma (inspectorista asetettava) fallback
        if (!string.IsNullOrEmpty(_fallbackDefaultLevelName))
            return _fallbackDefaultLevelName;

        // 4) Ei keksitty mitään
        Debug.LogError("[NetLevelLoader] ResolveDefaultLevelName() epäonnistui: ei current/default/fallback-nimeä.");
        return null;
    }

    public int IndexOfScene(string sceneName)
    {
        if (catalog == null) return -1;
        return catalog.IndexOfScene(sceneName); // LevelCatalogissa on tämä valmiina
    }

    // 2) Oletusindeksin ratkaisu ilman defaultIndex-kenttää
    public int ResolveDefaultIndex()
    {
        // Jos katalogi puuttuu/tyhjä → 0
        if (catalog == null || catalog.Count == 0) return 0;
```

```
        // a) Yritä LevelLoaderin oletusnimeä
        if (LevelLoader.Instance && !string.IsNullOrEmpty(LevelLoader.Instance.DefaultLevel))
        {
            int idx = catalog.IndexOfScene(LevelLoader.Instance.DefaultLevel);
            if (idx >= 0) return idx;
        }

        // b) Jos _currentLevel on asetettu (esim. aiemmasta pelistä)
        if (!string.IsNullOrEmpty(_currentLevel))
        {
            int idx = catalog.IndexOfScene(_currentLevel);
            if (idx >= 0) return idx;
        }

        // c) Fallback: katalogin ensimmäinen
        return 0;
    }
}
```

Assets/scripts/Oneline/LevelAndScenes/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/LevelAndScenes/NetWorkSceneBinder.cs

```
using System.Collections;
using Mirror;
using UnityEngine;
using UnityEngine.SceneManagement;
/// <summary>
///  Tämä pitää huolen siitä että kaikki objektit luodaan clientille oikeaan Level skeneen, kun client liittyy peliin.
///  Ilman tätä kaikki luotavat objektit latautusivat Coreen, mikä ei ole toivottavaa.
///  Käytä tätä kaikissa sellaisissa objekteissa jotka täytyy kentän latautuessa siirtyä oikeaan sceneen.
///  Esim Unit, Tuhoutuvat objektit jne.
///  Huom! SpawnRouter hoitaa objekti siirron oikeaan sceneen varsinainen pelin aikana.
/// </summary>
[DisallowMultipleComponent]
public class NetworkSceneBinder : NetworkBehaviour
{
    [SyncVar] public string targetSceneName;  // server täyttää tämän ennen spawnia

    public override void OnStartClient()
    {
        base.OnStartClient();
        // Hostilla server on jo siirtänyt oikein – ei tarvetta uudelleen siirtoon
        if (isServer) return;

        if (!string.IsNullOrEmpty(targetSceneName))
            StartCoroutine(MoveWhenLoaded(targetSceneName));
    }

    private IEnumerator MoveWhenLoaded(string sceneName)
    {
        // odota että kohdescene on varmasti ladattu clientillä
        while (true)
        {
            var sc = SceneManager.GetSceneByName(sceneName);
            if (sc.IsValid() && sc.isLoaded)
            {
                SceneManager.MoveGameObjectToScene(gameObject, sc);
                yield break;
            }
            yield return null;
        }
    }
}
```

Assets/scripts/Oneline/NetStaticHelpers/ActorIdUtil.cs

```
using Mirror;
using UnityEngine;

/// <summary>
/// Yksinkertainen, yhteinen paikka hakea "actorin" netId.
/// Käyttö: this.GetActorId() MISTÄ tahansa MonoBehaviourista.
/// </summary>
public static class ActorIdUtil
{
    /// <summary>
    /// Hakee lähimmän yläjuuren NetworkIdentityn ja palauttaa sen netId:n.
    /// Käy kaikissa: Actionit, Animatorit, ym. joissa 'this' on Component.
    /// </summary>
    public static uint GetActorId(this Component self)
    {
        if (!self) return 0;
        var ni = self.GetComponentInParent<NetworkIdentity>();
        return ni ? ni.netId : 0;
    }

    /// <summary>
    /// Jos haluat hakea suoraan Unitista (jos se on saatavilla).
    /// </summary>
    public static uint GetActorId(this Unit unit)
    {
        if (!unit) return 0;
        var ni = unit.GetComponent<NetworkIdentity>();
        return ni ? ni.netId : 0;
    }

    /// <summary>
    /// Variaatiot, jos joskus tarvitset GameObjectista tai Transformista.
    /// </summary>
    public static uint GetActorId(this GameObject go) => go ? go.transform.GetActorId() : 0;
    public static uint GetActorId(this Transform t)
    {
        if (!t) return 0;
        var ni = t.GetComponentInParent<NetworkIdentity>();
        return ni ? ni.netId : 0;
    }
}
```

Assets/scripts/Oneline/NetStaticHelpers/NetMode.cs

```
public static class NetMode
{
    public static bool IsOnline        => Mirror.NetworkServer.active || Mirror.NetworkClient.active;
    public static bool IsServer        => Mirror.NetworkServer.active;             // host tai dedicated
    public static bool IsClient        => Mirror.NetworkClient.active;             // host + remote client
    public static bool IsHost          => Mirror.NetworkServer.active && Mirror.NetworkClient.active;
    public static bool IsRemoteClient  => Mirror.NetworkClient.active && !Mirror.NetworkServer.active;
    public static bool IsDedicatedServer => Mirror.NetworkServer.active && !Mirror.NetworkClient.active;
    public static bool ServerOrOff => Mirror.NetworkServer.active || !Mirror.NetworkClient.isConnected; // Server or offline.
    public static bool Offline => !Mirror.NetworkClient.active && !Mirror.NetworkServer.active; // offline

}
```

Assets/scripts/Oneline/Secure/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/Sync/NetworkSync.cs

```
using Mirror;
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
{
    // --- Perus rooliliput (yhdessä paikassa) ---
    public static bool IsServer           => NetworkServer.active;
    public static bool IsClient           => NetworkClient.active;
    public static bool IsHost             => NetworkServer.active && NetworkClient.active;
    public static bool IsClientOnly       => !NetworkServer.active && NetworkClient.active;
    public static bool IsDedicatedServer  => NetworkServer.active && !NetworkClient.active;
    public static bool IsOffline => !NetworkServer.active && !NetworkClient.active;


    /// <summary>
    /// Hae NetworkClient netId:llä (toimii sekä clientillä että serverillä).
    /// Palauttaa null jos ei löydy (esim. ei vielä spawnattu tällä framella).
    /// </summary>
    public static NetworkIdentity FindIdentity(uint actorId)
    {
        if (actorId == 0) return null;
        NetworkClient.spawned.TryGetValue(actorId, out var ni);
        return ni;
    }
    /// <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, bool shouldHitUnits, uint actorNetId)
    {
        if (NetworkServer.active) // Online: server or host
        {
            Transform src = null;
            if (NetworkServer.spawned.TryGetValue(actorNetId, out var srcNI) && srcNI != null)
                src = srcNI.transform;

            SpawnRouter.SpawnNetworkServer(
                bulletPrefab, spawnPos, Quaternion.identity,
                source: src,
                sceneName: null,
                parent: null,
                owner: null,
                beforeSpawn: go =>
                {
                    if (go.TryGetComponent<BulletProjectile>(out var gp))
                    {
                        gp.actorUnitNetId = actorNetId;
                        gp.Setup(targetPos, shouldHitUnits);
                    }
                });

            return;
        }

        if (NetworkClient.active && NetworkSyncAgent.Local != null) // Online: client
        {
            NetworkSyncAgent.Local.CmdSpawnBullet(actorNetId, targetPos, shouldHitUnits);
        }
    }

    // HUOM: käytä tätä myös AE:stä (UnitAnimatorista)
    public static void SpawnGrenade(GameObject grenadePrefab, Vector3 spawnPos, Vector3 targetPos, float maxRangeWU, uint actorNetId)
    {

        if (NetworkServer.active) // Online: server tai host
        {
            Transform src = null;
            if (NetworkServer.spawned.TryGetValue(actorNetId, out var srcNI) && srcNI != null)
                src = srcNI.transform;

            SpawnRouter.SpawnNetworkServer(
                grenadePrefab, spawnPos, Quaternion.identity,
                source: src,            // löydetty actorNetId:llä
                sceneName: null,
```

```
                parent: null,
                owner: null,
                beforeSpawn: go =>
                {

                    var unit = src ? src.GetComponent<Unit>() : null;
                    int teamId = unit ? unit.GetTeamID() : -1;

                    if (go.TryGetComponent<GrenadeProjectile>(out var gp)) {
                        gp.actorUnitNetId = actorNetId;
                        gp.ownerTeamId = teamId;
                        gp.Setup(targetPos, maxRangeWU);
                    }
                });

            return;

        }

        if (NetworkClient.active && NetworkSyncAgent.Local != null) // Online: client
        {
            NetworkSyncAgent.Local.CmdSpawnGrenade(actorNetId, 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 unit, int amount, Vector3 hitPosition, uint actorNetId)
    {

        if (unit == null) return;

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

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

        if (NetworkClient.active) // Online: client
        {
            var ni = unit.GetComponent<NetworkIdentity>();
            if (ni && NetworkSyncAgent.Local != null)
            {
                if (unit == null || unit.IsDying() || unit.IsDead()) return;
```

```
                NetworkSyncAgent.Local.CmdApplyDamage(actorNetId, ni.netId, amount, hitPosition);
                return;
            }
        }

        unit.GetComponent<HealthSystem>().Damage(amount, hitPosition);
    }

    public static void ApplyDamageToObject(DestructibleObject target, int amount, Vector3 hitPosition, uint actorNetId)
    {
        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(actorNetId,ni.netId, amount, hitPosition);
                return;
            }
        }

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

    private static void UpdateHealthBarUI(HealthSystem healthSystem, Unit unit)
    {

        if (unit == null || healthSystem == null) return;
        // → 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(unit, healthSystem.GetHealth(), healthSystem.GetHealthMax());
            return;
        }

        if (NetworkClient.active && NetworkSyncAgent.Local != null)
        {
            if (unit == null || unit.IsDying() || unit.IsDead()) return;
            var ni = unit.GetComponent<NetworkIdentity>();
            if (ni != null)
                NetworkSyncAgent.Local.CmdRequestHpRefresh(ni.netId);
```

```
        }
    }

    public static void UpdateCoverUI(Unit unit)
    {
        if (unit == null || unit.IsDying() || unit.IsDead()) return;

        if (NetworkServer.active)
        {
            var agent = UnityEngine.Object.FindFirstObjectByType<NetworkSyncAgent>();
            if (agent != null)
                agent.ServerBroadcastCover(unit, unit.GetPersonalCover(), unit.GetPersonalCoverMax());
            return;
        }

        if (NetworkClient.active && NetworkSyncAgent.Local != null)
        {
            var ni = unit.GetComponent<NetworkIdentity>();
            if (ni != null)
                NetworkSyncAgent.Local.CmdRequestCoverRefresh(ni.netId);
        }
    }

    /// <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 || unit.IsDying() || unit.IsDead()) return;

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

        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, Vector3 lastHitPosition, int overkill)
    {

        if (NetworkServer.active)
```

```
        {
            // 1) Hae kaatuneen unitin Transform serveriltä netId:llä
            Transform src = null;
            if (NetworkServer.spawned.TryGetValue(sourceUnitNetId, out var srcNI) && srcNI != null)
                src = srcNI.transform;

            // 2) Spawnaa verkossa: siirto oikeaan level-sceneen hoituu SpawnRouterissa
            SpawnRouter.SpawnNetworkServer(
                prefab, pos, rot,
                source: src,          // → sama scene kuin kaatuneella unitilla
                sceneName: null,
                parent: null,
                owner: null,
                beforeSpawn: go =>
                {
                    if (go.TryGetComponent<UnitRagdoll>(out var rg))
                    {
                        rg.SetOverkill(overkill);
                        rg.SetLastHitPosition(lastHitPosition);
                    }
                    if (go.TryGetComponent<RagdollPoseBinder>(out var binder))
                    {
                        binder.sourceUnitNetId = sourceUnitNetId;
                        binder.lastHitPos = lastHitPosition;
                        binder.overkill = overkill;
                    }
                    else
                    {
                        Debug.LogWarning("[Ragdoll] Ragdoll prefab lacks RagdollPoseBinder.");
                    }
                });

            return;
        }
    }

    public static bool IsOwnerHost(uint ownerId)
    {
        if (!NetworkServer.active) return false; // varmin tieto vain serverillä
        foreach (var kv in NetworkServer.connections)
        {
            var conn = kv.Value;
            if (conn?.identity && conn.identity.netId == ownerId)
                return conn.connectionId == 0; // 0 = host
        }
        return false;
    }

    /// <summary>Onko tämä NetworkIdentity omistettu tällä koneella?</summary>
    public static bool IsOwnedHere(NetworkIdentity ni)
        => ni != null && (ni.isOwned || ni.isLocalPlayer);
```

```
    /// <summary>
    /// Paikallisen pelaajan teamId (piirtäjät, HUD, yms. käyttävät tätä).
    /// - Offline: 0
    /// - Versus: host=0, puhdas client=1
    /// - SP/Coop online: 0
    /// </summary>
    public static int GetLocalPlayerTeamId(GameMode mode)
    {
        if (IsOffline) return 0;

        if (mode == GameMode.Versus) return IsServer ? 0 : 1;
        return 0;
    }

    public static void TriggerOverwatchShot(Unit watcher, Unit target, GridPosition targetGridPos)
    {

        if (watcher == null || target == null || watcher.IsDead() || watcher.IsDying())
        {
            return;
        }

        var shoot = watcher.GetComponent<ShootAction>();
        if (shoot == null)
        {
            Debug.LogWarning($"[OW-NetworkSync] Watcher {watcher.name} has no ShootAction!");
            return;
        }

        if (!NetworkServer.active && !NetworkClient.active)
        {
            shoot.MarkAsOverwatchShot(true);
            shoot.TakeAction(targetGridPos, () => { });
            return;
        }

        if (NetworkServer.active)
        {
            shoot.MarkAsOverwatchShot(true);
            shoot.TakeAction(targetGridPos, () => { });

            var agent = UnityEngine.Object.FindFirstObjectByType<NetworkSyncAgent>();
            if (agent != null)
            {
                var watcherNi = watcher.GetComponent<NetworkIdentity>();
                if (watcherNi != null)
                {

                    agent.ServerBroadcastOverwatchShot(watcherNi.netId, targetGridPos.x, targetGridPos.z, targetGridPos.floor);
                   // agent.ServerFlashOverwatchCone(watcherNi.netId);
                }
            }
```

```
            return;
        }

        Debug.LogWarning("[OW-NetworkSync] Client-only mode should not call TriggerOverwatchShot!");
    }

}
```

Assets/scripts/Oneline/Sync/NetworkSyncAgent.cs

```
using System.Collections;
using System.Collections.Generic;
using Mirror;
using UnityEngine;
/// <summary>
/// Attached to the PlayerController GameObject.
/// 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.
///
/// 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
    public override void OnStartAuthority() { Local = this; }
    public override void OnStopAuthority() { if (Local == this) Local = null; }

    [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="clientSuggestedTarget">World position the bullet is travelling towards.</param>

    [Command(requiresAuthority = true)]
    public void CmdSpawnBullet(uint actorNetId, Vector3 clientSuggestedTarget, bool shouldHitUnits)
    {
        if (!NetworkServer.active) return;
        if (bulletPrefab == null) { Debug.LogWarning("[NetSyncAgent] bulletPrefab missing"); return; }
        if (actorNetId == 0 || !RightOwner(actorNetId)) return;

        if (!NetworkServer.spawned.TryGetValue(actorNetId, out var actorNi) || actorNi == null) return;

        var ua = actorNi.GetComponent<UnitAnimator>();
        Vector3 origin = (ua && ua.ShootPoint) ? ua.ShootPoint.position : actorNi.transform.position;
        Vector3 target = clientSuggestedTarget;
```

```
        SpawnRouter.SpawnNetworkServer(
            bulletPrefab, origin, Quaternion.identity,
            source: actorNi.transform,
            sceneName: null,
            parent: null,
            owner: connectionToClient,
            beforeSpawn: go =>
            {
                if (go.TryGetComponent<BulletProjectile>(out var bp))
                {
                    bp.actorUnitNetId = actorNetId;
                    bp.Setup(target, shouldHitUnits);
                }
            });
    }

    [Command(requiresAuthority = true)]
    public void CmdSpawnGrenade(uint actorNetId, Vector3 clientSuggestedTarget)
    {
        if (!NetworkServer.active) return;
        if (grenadePrefab == null) { Debug.LogWarning("[NetSyncAgent] GrenadePrefab missing"); return; }
        if (actorNetId == 0 || !RightOwner(actorNetId)) return;

        if (!NetworkServer.spawned.TryGetValue(actorNetId, out var actorNi) || actorNi == null) return;

        var ua = actorNi.GetComponent<UnitAnimator>();
        if (!ua || !ua.ThrowPoint) return;

        Vector3 origin = ua.ThrowPoint.position;
        Vector3 target = clientSuggestedTarget;

        SpawnRouter.SpawnNetworkServer(
            grenadePrefab, origin, Quaternion.identity,
            source: actorNi.transform,
            sceneName: null,
            parent: null,
            owner: connectionToClient,
            beforeSpawn: go =>
            {
                var unit = actorNi.GetComponent<Unit>();
                int teamId = unit ? unit.GetTeamID() : -1;

                if (go.TryGetComponent<GrenadeProjectile>(out var gp))
                {
                    gp.actorUnitNetId = actorNetId;
                    gp.ownerTeamId = teamId;
                    gp.Setup(target);
                }
            });
    }

    private bool RightOwner(uint actorNetId)
```

```
    {
        // Varmista että soittaja omistaa kyseisen actor-yksikön
        if (!NetworkServer.spawned.TryGetValue(actorNetId, out var actorIdentity) || actorIdentity == null) return false;

        // actorin todellinen omistaja.
        var actorUnit = actorIdentity.GetComponent<Unit>();

        // Client joka lähetti comennon.
        var callerOwnerId = connectionToClient.identity?.netId ?? 0;

        // Unitissa on OwnerId, jonka asetus tehdään spawnaaessa (SpawnUnitsCoordinator) → käytä sitä checkissä
        // OwnerId = PlayerController-objektin netId
        if (actorUnit == null || actorUnit.OwnerId != callerOwnerId) return false;

        return true;
    }

    /// <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 actorNetId, uint targetNetId, int amount, Vector3 hitPosition)
    {

        if (!NetworkServer.spawned.TryGetValue(targetNetId, out var targetNi) || targetNi == null) return;
        if (!RightOwner(actorNetId)) return;

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

        // --- NEW: server-side sanity cap ---
        int maxAllowed = 0;
        if (NetworkServer.spawned.TryGetValue(actorNetId, out var attackerNi) && attackerNi != null)
        {
            var attacker = attackerNi.GetComponent<Unit>();
            var w = attacker != null ? attacker.GetCurrentWeapon() : null;

            // Aseesta johdettu maksimi: Miss/Graze/Hit/Crit → enintään base + critBonus
            if (w != null)
                maxAllowed = Mathf.Max(maxAllowed, w.baseDamage + w.critBonusDamage); // esim. 10 + 8, jne.

            // Lähitaistelulle varmuuskatto (sinulla MeleeAction.damage = 100 → ota vähintään tämä)
            // Vältetään riippuvuus MeleeActionin yksityiseen kenttään ottamalla varovainen fallback:
            maxAllowed = Mathf.Max(maxAllowed, 100); // MeleeActionissa serialize'd damage=100. :contentReference[oaicite:2]{index=2}
        }

        int safe = Mathf.Clamp(amount, 0, maxAllowed);
        if (safe != amount)
            Debug.LogWarning($"[Server] Clamped damage from {amount} to {safe} (actor {actorNetId} → target {targetNetId}).");
```

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

        // 2) Broadcast UI (kuten ennenkin)
        ServerBroadcastHp(unit, hs.GetHealth(), hs.GetHealthMax());
    }

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

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

        obj.Damage(amount, hitPosition);
    }

    [Command(requiresAuthority = false)]
    public void CmdRequestHpRefresh(uint unitNetId)
    {
        if (!NetworkServer.active) return;
        if (!NetworkServer.spawned.TryGetValue(unitNetId, out var id)) return;

        var u = id.GetComponent<Unit>();
        var hs = u ? u.GetComponent<HealthSystem>() : null;
        if (u == null || hs == null) return;

        ServerBroadcastHp(u, hs.GetHealth(), hs.GetHealthMax()); // server lukee
    }

    // ---- 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]
    public void ServerBroadcastCover(Unit unit, int current, int max)
    {
        var ni = unit.GetComponent<NetworkIdentity>();
```

```
        if (ni) RpcNotifyCoverChanged(ni.netId, current, max);
    }

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

        var unit = id.GetComponent<Unit>();
        if (unit == null) return;

        unit.ApplyNetworkCover(current, max);
    }

    [Command(requiresAuthority = false)]
    public void CmdRequestCoverRefresh(uint unitNetId)
    {
        if (!NetworkServer.spawned.TryGetValue(unitNetId, out var id) || id == null) return;
        var unit = id.GetComponent<Unit>();
        if (unit == null) return;

        // Server lukee arvot ja broadcastaa
        ServerBroadcastCover(unit, unit.GetPersonalCover(), unit.GetPersonalCoverMax());
    }

    [Command(requiresAuthority = false)]
    public void CmdSetUnitCover(uint unitNetId, int value)
    {
        if (!NetworkServer.spawned.TryGetValue(unitNetId, out var id) || id == null) return;
        var unit = id.GetComponent<Unit>();
        if (!unit) return;

        unit.SetPersonalCover(Mathf.Clamp(value, 0, unit.GetPersonalCoverMax()));
    }

    [Command(requiresAuthority = false)]
    public void CmdSetUnderFire(uint unitNetId, bool value)
    {

        if (!NetworkServer.spawned.TryGetValue(unitNetId, out var id) || id == null) return;
        var unit = id.GetComponent<Unit>();
        if (!unit) return;

        unit.SetUnderFireServer(value);
    }

    [Server]
    public void ServerBroadcastUnderFire(Unit unit, bool value)
    {
        var ni = unit.GetComponent<NetworkIdentity>();
        if (ni) RpcNotifyUnderFireChanged(ni.netId, value);
```

```
    }

    [ClientRpc]
    void RpcNotifyUnderFireChanged(uint unitNetId, bool value)
    {
        if (!NetworkClient.spawned.TryGetValue(unitNetId, out var id) || id == null) return;
        var unit = id.GetComponent<Unit>();
        if (!unit) return;

        unit.ApplyNetworkUnderFire(value);
    }

    // ---- 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
    }

    [Command]
    public void CmdRegenCoverOnMove(uint unitNetId, int distance)
    {
        if (!NetworkServer.spawned.TryGetValue(unitNetId, out NetworkIdentity ni)) return;
        var cs = ni.GetComponent<CoverSkill>();
        if (cs != null) cs.ServerRegenCoverOnMove(distance);
```

```
    }
    [Command]
    public void CmdResetCurrentCoverBonus(uint unitNetId)
    {
        if (!NetworkServer.spawned.TryGetValue(unitNetId, out NetworkIdentity ni)) return;
        var cs = ni.GetComponent<CoverSkill>();
        if (cs != null) cs.ServerResetCurrentCoverBonus();
    }

    [Command(requiresAuthority = false)]
    public void CmdApplyCoverBonus(uint unitNetId)
    {

        if (!NetworkServer.spawned.TryGetValue(unitNetId, out NetworkIdentity ni)) return;
        var cs = ni.GetComponent<CoverSkill>(); // tai GetComponentInChildren<CoverSkill>()
        if (cs != null) cs.ServerApplyCoverBonus();
    }

    [Server]
    public void ServerBroadcastOverwatchShot(uint watcherNetId, int targetX, int targetZ, int targetFloor)
    {
        RpcExecuteOverwatchShot(watcherNetId, targetX, targetZ, targetFloor);
    }

    [ClientRpc]
    void RpcExecuteOverwatchShot(uint watcherNetId, int targetX, int targetZ, int targetFloor)
    {
        if (!NetworkClient.spawned.TryGetValue(watcherNetId, out var watcherNi) || watcherNi == null)
        {
            Debug.LogWarning($"[OW-Client] Watcher netId {watcherNetId} not found in spawned objects!");
            return;
        }

        var watcher = watcherNi.GetComponent<Unit>();
        if (watcher == null)
        {
            Debug.LogWarning($"[OW-Client] Watcher has no Unit component!");
            return;
        }

        if (watcher.IsDead() || watcher.IsDying())
        {
            Debug.LogWarning($"[OW-Client] Watcher {watcher.name} is dead or dying!");
            return;
        }

        var shoot = watcher.GetComponent<ShootAction>();
        if (shoot == null)
        {
            Debug.LogWarning($"[OW-Client] Watcher {watcher.name} has no ShootAction!");
            return;
        }
```

```
        GridPosition targetGridPos = new GridPosition(targetX, targetZ, targetFloor);

        var targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(targetGridPos);
        if (targetUnit == null)
        {
            Debug.LogWarning($"[OW-Client] No target unit found at {targetGridPos}! Shot cancelled.");
            return;
        }

        shoot.MarkAsOverwatchShot(true);
        shoot.TakeAction(targetGridPos, () => { });
    }

    [Command(requiresAuthority = false)]
    public void CmdCheckOverwatchStep(uint unitNetId, int gridX, int gridZ, int gridFloor)
    {
        if (!NetworkServer.active) return;

        if (!NetworkServer.spawned.TryGetValue(unitNetId, out var unitNi) || unitNi == null) return;

        var unit = unitNi.GetComponent<Unit>();
        if (unit == null || unit.IsDead() || unit.IsDying()) return;

        GridPosition gridPos = new GridPosition(gridX, gridZ, gridFloor);
        StatusCoordinator.Instance.CheckOverwatchStep(unit, gridPos);
    }


    [ClientRpc]
    public void RpcRebuildTeamVision(uint[] unitNetIds, bool endPhase)
    {
        foreach (var id in unitNetIds)
        {
            if (!Mirror.NetworkClient.spawned.TryGetValue(id, out var ni) || ni == null) continue;

            var u = ni.GetComponent<Unit>();
            var v = u ? u.GetComponent<UnitVision>() : null;
            if (u == null || v == null || !v.IsInitialized) continue;

            // 1) aina tuore 360° cache
            v.UpdateVisionNow();

            Vector3 facing = u.transform.forward;
            facing.y = 0f;
            float angle = 360f;

            // 2) ENSIN OverwatchPayload (serveriltä päivitetty)
            if (u.TryGetComponent<UnitStatusController>(out var status) &&
                status.TryGet<OverwatchPayload>(UnitStatusType.Overwatch, out var payload))
            {
                facing = (payload.facingWorld.sqrMagnitude > 1e-6f) ? payload.facingWorld : facing;
```

```
            angle = payload.coneAngleDeg;            // esim. 80f
        }
        // 3) Muuten jos endPhase ja OW päällä -> johda suunnasta
        else if (endPhase && u.TryGetComponent<OverwatchAction>(out var ow) && ow.IsOverwatch())
        {
            var dir = ow.TargetWorld - u.transform.position; dir.y = 0f;
            if (dir.sqrMagnitude > 1e-4f) facing = dir.normalized;
            angle = 80f;
        }
        // 4) Muuten fallback AP-logiikkaan
        else
        {
            angle = endPhase ? v.VisionPenaltyWhenUsingAP(u.GetActionPoints())
                : 360f;
        }

        // 5) julkaisu
        v.ApplyAndPublishDirectionalVision(facing, angle);


    }
}

[ClientRpc]
public void RpcUpdateSingleUnitVision(uint unitNetId, Vector3 facing, float coneAngle)
{
    if (!Mirror.NetworkClient.spawned.TryGetValue(unitNetId, out var ni) || ni == null) return;

    var u = ni.GetComponent<Unit>();
    var v = u ? u.GetComponent<UnitVision>() : null;
    if (u == null || v == null || !v.IsInitialized) return;

    // Päivitä tämän unitin 360° cache
    v.UpdateVisionNow();
    // Päivitä visuaalinen näkymä.
    v.ShowUnitOverWatchVision(facing, coneAngle);
    // Julkaise uusi kartio
    v.ApplyAndPublishDirectionalVision(facing, coneAngle);
}

[Server]
public void ServerPushUnitVision(uint unitNetId, float fx, float fz, float coneDeg)
{
    // hostille (serverin oma näkymä) päivitys heti paikallisesti:
    if (Mirror.NetworkServer.active)
        RebuildUnitVisionLocal(unitNetId, fx, fz, coneDeg);

    // lähetä myös clienteille
    RpcRebuildUnitVision(unitNetId, fx, fz, coneDeg);
}

[ClientRpc]
```

```
    private void RpcRebuildUnitVision(uint unitNetId, float fx, float fz, float coneDeg)
    {
        RebuildUnitVisionLocal(unitNetId, fx, fz, coneDeg);
    }

    // Sama koodi serverille ja clientille: EI kutsuta UpdateVisionNow()
    // → ei mitään 360° välikuvia. Julkaistaan suoraan kartio.
    private static void RebuildUnitVisionLocal(uint unitNetId, float fx, float fz, float coneDeg)
    {
        if (!Mirror.NetworkClient.spawned.TryGetValue(unitNetId, out var ni) || ni == null) return;
        var u = ni.GetComponent<Unit>();
        var v = ni.GetComponent<UnitVision>();
        if (u == null || v == null || !v.IsInitialized) return;

        var facing = new Vector3(fx, 0f, fz);
        // EI kutsuta v.UpdateVisionNow() — käytetään olemassa olevaa cachea
        v.ApplyAndPublishDirectionalVision(facing, coneDeg);

        // halutessasi päivitä myös OW-overlay
        v.ShowUnitOverWatchVision(facing, coneDeg);
    }

    [Server]
    public void ServerPushTeamVision(int teamId, bool endPhase)
    {
        var ids = CollectTeamUnitIds(teamId);
        RpcRebuildTeamVision(ids, endPhase);
    }

    [Server]
    private static uint[] CollectTeamUnitIds(int teamId)
    {
        var um = UnitManager.Instance;
        var list = um != null ? um.GetAllUnitList() : null;
        var ids = new List<uint>();
        if (list == null) return ids.ToArray();

        foreach (var unit in list)
        {
            if (!unit) continue;

            int t = unit.GetTeamID();
            if (t != teamId) continue;
            if (unit.IsDying() || unit.IsDead()) continue;

            var ni = unit.GetComponent<NetworkIdentity>();
            if (ni != null) ids.Add(ni.netId);
        }
        return ids.ToArray();
    }

    [Command(requiresAuthority = false)]
```

```
    public void CmdSetOverwatch(uint unitNetId, bool value, float fx, float fz)
    {
        if (!Mirror.NetworkServer.spawned.TryGetValue(unitNetId, out var ni) || ni == null) return;

        var u = ni.GetComponent<Unit>();
        var ow = ni.GetComponent<OverwatchAction>();
        if (!u || !ow) return;

        ow.ServerApplyOverwatch(value, new Vector2(fx, fz));
    }

    [ClientRpc]
    public void RpcClearAllOverwatchVisuals(int teamID)
    {
        if (GridSystemVisual.Instance == null) return;
        // Poista vain oman tiimin overwatch-visuaalit
        GridSystemVisual.Instance.ClearTeamOverwatchVisuals(teamID);
    }

    [Command(requiresAuthority = false)]
    public void CmdUpdateOverwatchCone(uint unitNetId, float fx, float fz, float coneAngle)
    {
        if (!Mirror.NetworkServer.spawned.TryGetValue(unitNetId, out var ni) || ni == null) return;

        var u = ni.GetComponent<Unit>();
        var v = u ? u.GetComponent<UnitVision>() : null;
        if (u == null || v == null || !v.IsInitialized) return;

        var facing = new Vector3(fx, 0f, fz);
        facing.Normalize();

        if (u.TryGetComponent<UnitStatusController>(out var status))
        {
            if (status.TryGet<OverwatchPayload>(UnitStatusType.Overwatch, out var payload))
            {
                payload.facingWorld = facing;
                payload.coneAngleDeg = coneAngle;
                status.AddOrUpdate(UnitStatusType.Overwatch, payload);
            }
        }

        v.UpdateVisionNow();
        v.ShowUnitOverWatchVision(facing, coneAngle);
        v.ApplyAndPublishDirectionalVision(facing, coneAngle);

        RpcUpdateSingleUnitVision(unitNetId, facing, coneAngle);
    }

}
```

Assets/scripts/Oneline/UI/RelayJoinCodeUI.cs

```
using UnityEngine;
using TMPro;

[DefaultExecutionOrder(-100)]
public class RelayJoinCodeUI : MonoBehaviour
{
    public static RelayJoinCodeUI Instance { get; private set; }

    [SerializeField] private GameObject root;      // vedä tähän CodeCanvas TAI paneelin juuri
    [SerializeField] private TMP_Text codeText;    // vedä JoinCodeText


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

        if (root == null) root = gameObject;       // fallback: käytä CodeCanvasia juurena
        DontDestroyOnLoad(root);                   // pysyy scene-vaihdon yli
        if (root.activeSelf) root.SetActive(false);
        //Hide();
    }

    // Turvahaku siltä varalta, että Instance ei ole vielä asetettu
    public static RelayJoinCodeUI GetOrFind()
    {
        if (Instance != null) return Instance;
        var found = FindFirstObjectByType<RelayJoinCodeUI>(FindObjectsInactive.Include);
        if (found != null)
        {
            Instance = found;
            if (found.root == null) found.root = found.gameObject;
        }
        return Instance;
    }

    public void ShowCode(string code)
    {
        var c = (code ?? "").Trim().ToUpperInvariant();
        if (codeText) codeText.text = $"JOIN CODE: {c}";
        root.SetActive(true);
    }

    public void Hide() => root.SetActive(false);

}
```

Assets/scripts/Oneline/UnitOneline/DeathStopper.cs

```
using UnityEngine;
using Mirror;

/// <summary>
/// DeathStopper: varma "hätäjarru" kun Unit kuolee kesken liikkeen verkossa.
/// Kutsu serverillä ENNEN FreezeBeforeDespawn():ia.
/// Toiminnot:
/// 1) Pysäyttää MoveActionin deterministisesti (ForceStopNow -> ActionComplete -> UI busy vapautuu).
/// 2) Lähettää ClientRpc-teleportin samaan paikkaan (snäppää ja nollaa interpolaatiopuskurin).
/// 3) Nollaa mahdollisen rigidbodyn nopeudet.
/// 4) (Valinn.) kytkee *minkä tahansa* NetworkTransformin pois/päälle clientilla ilman kovaa tyyppiriippuvuutta.
///    -> Ei tarvitse viitata Mirror.Components -asmdeffiin, joten tämä kääntyy vaikka et lisää sitä.
/// </summary>
[RequireComponent(typeof(NetworkIdentity))]
public class DeathStopper : NetworkBehaviour
{
    [Header("Optional component refs (autofill, jos tyhjä)")]
    [SerializeField] private MoveAction moveAction;

    // HUOM: Ei kovaa viittausta NetworkTransform / NetworkAnimator -tyyppeihin.
    // Haetaan ne niminä, jolloin tämä skripti ei vaadi Mirror.Components -asmdef -referenssiä.
    [SerializeField] private Behaviour netTransformBehaviour; // esim. "NetworkTransform"
    [SerializeField] private Behaviour netAnimatorBehaviour;  // esim. "NetworkAnimator"

    [SerializeField] private Animator animator;
    [SerializeField] private Rigidbody rb;

    [Header("Behavior")]
    [Tooltip("Kytke NT pois/päälle clientilla snäpin yhteydessä, jotta interpolaatiopuskurit nollautuvat.")]
    [SerializeField] private bool toggleNetworkTransformOnSnap = true;

    void Awake()
    {
        if (!moveAction) moveAction = GetComponent<MoveAction>();

        // Etsi dynaamisesti nimellä -> ei tarvita tyyppiviitettä
        if (!netTransformBehaviour)
            netTransformBehaviour = GetComponent("NetworkTransform") as Behaviour;
        if (!netAnimatorBehaviour)
            netAnimatorBehaviour = GetComponent("NetworkAnimator") as Behaviour;

        if (!animator) animator = GetComponentInChildren<Animator>();
        if (!rb) rb = GetComponent<Rigidbody>();
    }

    /// <summary>
    /// SERVER ONLY. Katkaise liike, snäppää pose kaikille ja valmistele tuho.
    /// Kutsu tämä heti, kun päätät että unit kuolee (ennen komponenttien disablointia).
    /// </summary>
/*
```

```
    [Server]
    public void HaltForDeath()
    {
        // 1) Pysäytä liike deterministisesti (ennen kuin actionit disabloidaan)
        try
        {
            if (moveAction)
            {
                moveAction.ForceStopNow(); // kutsuu ActionComplete() -> UI busy vapautuu
            }
        }
        catch (System.Exception ex)
        {
            Debug.LogWarning($"[DeathStopper] ForceStopNow() poikkeus: {ex.Message}");
        }

        // 2) Snäppää nykyiseen paikkaan kaikki clientit (ja nollaa interpolaatiopuskuri)
        var pos = transform.position;
        var rot = transform.rotation;
        RpcTeleportSnap(pos, rot);
    }
    */

    /// <summary>
    /// Katkaise liike, snäppää pose kaikille ja valmistele tuho.
    /// Toimii sekä offline- että online-tilassa.
    /// </summary>
    // POISTA [Server]

    public void HaltForDeath()
    {
        // 1) Pysäytä liike deterministisesti (ennen kuin actionit disabloidaan)
        try
        {
            if (moveAction)
                moveAction.ForceStopNow(); // kutsuu ActionComplete() jne.
        }
        catch (System.Exception ex)
        {
            Debug.LogWarning($"[DeathStopper] ForceStopNow() poikkeus: {ex.Message}");
        }

        // 2) Snap: serverissä RPC; offline-lokisesti paikallinen snap
        var pos = transform.position;
        var rot = transform.rotation;

        if (NetworkServer.active)
        {
            // online/server-polku
            RpcTeleportSnap(pos, rot);   // [ClientRpc] metodi, ajetaan vain kun server aktiivinen
        }
        else
```

```
        {
            // offline-polku – EI RPC:tä
            SnapLocal(pos, rot);
        }
    }

    // Apumetodi offline-snappiin
    private void SnapLocal(Vector3 pos, Quaternion rot)
    {
        transform.SetPositionAndRotation(pos, rot);

        if (TryGetComponent<UnityEngine.AI.NavMeshAgent>(out var agent))
        {
            agent.ResetPath();
            agent.Warp(pos);
            transform.rotation = rot;
        }
    }

    [ClientRpc]
    private void RpcTeleportSnap(Vector3 pos, Quaternion rot)
    {

        try
        {
            // a) Katkaise hetkeksi (mahd.) NetworkTransform (tyhjentää interpolaation)
            bool reenableNT = false;
            if (toggleNetworkTransformOnSnap && netTransformBehaviour && netTransformBehaviour.enabled)
            {
                netTransformBehaviour.enabled = false;
                reenableNT = true;
            }

            // b) Aseta transform suoraan
            transform.SetPositionAndRotation(pos, rot);

            // c) Nollaa mahdollinen fysiikka-liike
            if (rb)
            {
                rb.linearVelocity = Vector3.zero;
                rb.angularVelocity = Vector3.zero;
                rb.Sleep();
            }

            // d) "Pingota" animaattori frameen (estää yhden framen juoksuanimaatiojäännöksen)
            if (animator) animator.Update(0f);

            // e) Ota (mahd.) NetworkTransform takaisin käyttöön
            if (reenableNT && netTransformBehaviour)
            {
                netTransformBehaviour.enabled = true;
            }
```

```
        }
        catch (System.Exception ex)
        {
            Debug.LogWarning($"[DeathStopper] RpcTeleportSnap poikkeus: {ex.Message}");
        }
    }

    /// <summary>
    /// Apumetodi: turvallinen yritys kutsua hätäjarru.
    /// </summary>
    [Server]
    public static void TryHalt(Unit unit)
    {
        if (!unit)
        {
            Debug.Log("[DeathStopper] Unit is null!");
            return;
        }
        var ds = unit.GetComponent<DeathStopper>();
        if (ds == null) return;

        // HaltForDeath itse voi sisäisesti vartioida server-jutut:
        ds.HaltForDeath(); // EI [Server]
    }

    public static void TryHaltOfline(Unit unit)
    {
        if (!unit)
        {
            Debug.Log("[DeathStopper] Unit is null!");
            return;
        }
        var ds = unit.GetComponent<DeathStopper>();
        if (ds == null) return;

        // HaltForDeath itse voi sisäisesti vartioida server-jutut:
        ds.HaltForDeath(); // EI [Server]
    }
}
```

Assets/scripts/Oneline/UnitOneline/ResetService.cs

```
using Mirror;
using UnityEngine;

public class ResetService : NetworkBehaviour
{
    public static ResetService Instance;
    void Awake() => Instance = this;

    /// <summary>
    /// Kutsu tätä Play Again -napista. Hoitaa online/offline-haarat.
    /// </summary>
    public void RequestReset()
    {
        if (NetworkServer.active)                // HOST / DEDISERVER
        {
            var win = FindFirstObjectByType<WinBattle>(FindObjectsInactive.Include);
            if (win != null) win.HideEndPanel();

            // UUSI: tyhjennä unit-listat ennen reloadia
            var um = FindFirstObjectByType<UnitManager>(FindObjectsInactive.Include);
            if (um) um.ClearAllUnitLists();

            NetTurnManager.Instance.ResetTurnState();
            TurnSystem.Instance.ResetTurnId();
            TurnSystem.Instance.ResetTurnNumber();

            RpcPreResetHud(); // siistii kaikkien HUDit
            NetLevelLoader.Instance.ServerReloadCurrentLevel();
            return;
        }

        if (NetworkClient.active)                // PUHDAS CLIENT
        {
            // Serveri hoitaa reloadin ja aloituksen
            return;
        }

        // OFFLINE
        LevelLoader.Instance.ReloadOffline(LevelLoader.Instance.DefaultLevel);
    }

    /// <summary>
    /// Kevyt ja turvallinen UI-siistintä: piilota end-panelit.
    /// </summary>
    [ClientRpc]
    void RpcPreResetHud()
    {
        TurnSystem.Instance.ResetTurnId();
        var win = FindFirstObjectByType<WinBattle>(FindObjectsInactive.Include);
        if (win != null) win.HideEndPanel();
```

```
        // UUSI: myös asiakkaan UnitManager nollaan
        var um = FindFirstObjectByType<UnitManager>(FindObjectsInactive.Include);
        if (um) um.ClearAllUnitLists();
    }
}
```

Assets/scripts/Oneline/VisibilitySync/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/VisibilitySync/NetworkVisionRpcHub.cs

```
using Mirror;
using UnityEngine;

public class NetworkVisionRpcHub : NetworkBehaviour
{
    public static NetworkVisionRpcHub Instance;
    void Awake() => Instance = this;

    [ClientRpc]
    public void RpcResetLocalVisionAndRebuild()
    {
        var tvs = TeamVisionService.Instance;
        if (tvs != null)
        {
            tvs.ClearTeamVision(0);
            tvs.ClearTeamVision(1);
        }

        foreach (var uv in Object.FindObjectsByType<UnitVision>(FindObjectsSortMode.None))
            if (uv != null && uv.IsInitialized)
                uv.UpdateVisionNow();
    }
}
```

Assets/scripts/Oneline/VisibilitySync/WeaponVisibilitySync.cs

```
using System;
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;

    private HealthSystem hs;
    private bool frozen;

    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>();
        TryGetComponent(out hs);
    }

    void OnEnable()
    {
        if (hs != null)
        {
            hs.OnDying += OnDying;
            hs.OnDead  += OnDead;
        }
    }
    void OnDisable()
    {
        if (hs != null)
        {
            hs.OnDying -= OnDying;
            hs.OnDead -= OnDead;
        }
    }

    private void OnDying(object s, EventArgs e) => frozen = true;
    private void OnDead (object s, EventArgs e) => frozen = true;

    // --- OWNER kutsuu tätä (esim. AE:ssä) ---
    public void OwnerRequestSet(bool rifleRight,bool rifleLeft, bool meleeLeft, bool grenade)
    {
```

```
        // ÄLÄ lähetä verkkoon jos kuolemassa/kuollut → juuri tämä poistaa varoitusspämmin
        if (frozen || (hs && (hs.IsDying() || hs.IsDead())))
            return;

        // 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)
    {
        if (frozen) return; // varmistetaan myös server-puolella
        // 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;
    }

    [Server]
    public void ServerForceSet(bool rifleRight, bool rifleLeft, bool meleeLeft, bool grenade)
    {
        // Aseta server-authoritatiivinen tila (jos käytät NetVisibilityä)
        if (rifleRightVis)   rifleRightVis.ServerSetVisible(rifleRight);
        if (rifleLeftVis)    rifleLeftVis.ServerSetVisible(rifleLeft);
        if (meleeLeftVis)    meleeLeftVis.ServerSetVisible(meleeLeft);
        if (grenadeRightVis) grenadeRightVis.ServerSetVisible(grenade);

        // Ja lähetä varmuuden vuoksi kaikille klienteille
        RpcSet(rifleRight, rifleLeft, meleeLeft, grenade);
    }
}
```

Assets/scripts/Systems/SpawnRouter.cs

```
// Assets/Scripts/Runtime/Systems/SpawnRouter.cs
using System;
using UnityEngine;
using UnityEngine.SceneManagement;
using Mirror;

public static class SpawnRouter
{
    /// <summary>Muokkaa tätä jos Core on eri nimellä.</summary>
    public static string CoreSceneName = "Core";

    /// <summary>
    /// Paikallinen spawn (ei NetworkServer.Spawn). Palauttaa instanssin.
    /// </summary>
    public static GameObject SpawnLocal(
        GameObject prefab,
        Vector3 pos,
        Quaternion rot,
        Transform source = null,
        string sceneName = null,
        Transform parent = null,
        Action<GameObject> beforeReturn = null)
    {
        var go = UnityEngine.Object.Instantiate(prefab, pos, rot);
        if (parent != null) go.transform.SetParent(parent, true);

        var targetScene = ResolveScene(source, sceneName);
        if (targetScene.IsValid()) SceneManager.MoveGameObjectToScene(go, targetScene);

        beforeReturn?.Invoke(go);
        return go;
    }

    /// <summary>
    /// Server-spawn (Mirror). Luo instanssin, siirtää sen level-scenelle ja kutsuu NetworkServer.Spawn.
    /// </summary>
    public static GameObject SpawnNetworkServer(
        GameObject prefab,
        Vector3 pos,
        Quaternion rot,
        Transform source = null,
        string sceneName = null,
        Transform parent = null,
        NetworkConnectionToClient owner = null,
        Action<GameObject> beforeSpawn = null)
    {
        if (!NetworkServer.active)
        {
            Debug.LogWarning("[SpawnRouter] SpawnNetworkServer() called without an active server.");
            return null;
```

```
        }

        var go = UnityEngine.Object.Instantiate(prefab, pos, rot);
        if (parent != null) go.transform.SetParent(parent, true);

        // 1) Siirrä instanssi oikeaan level-scenelle (host/servu puolella)
        var targetScene = ResolveScene(source, sceneName);
        if (targetScene.IsValid()) SceneManager.MoveGameObjectToScene(go, targetScene);

        // 2) Tee mahdolliset initit ennen spawnia
        beforeSpawn?.Invoke(go);

        // 3) (UUSI) Kerro myös clienteille mihin sceneen tämä kuuluu
        var binder = go.GetComponent<NetworkSceneBinder>();
        if (binder != null && targetScene.IsValid())
            binder.targetSceneName = targetScene.name;
    #if UNITY_EDITOR
        else
            Debug.LogWarning($"[SpawnRouter] Prefab '{prefab.name}' lacks NetworkSceneBinder. Client will keep it in Core.");
    #endif

        // 4) Verkkoon
        if (!go.TryGetComponent<NetworkIdentity>(out _))
            Debug.LogWarning("[SpawnRouter] Network spawn prefab has no NetworkIdentity.");

        if (owner != null) NetworkServer.Spawn(go, owner);
        else NetworkServer.Spawn(go);

        return go;
    }

    /// <summary>
    /// Yhtenäinen scene-resoluutio:
    /// 1) nimellä, 2) lähdeobjektin scene, 3) ensimmäinen ladattu ei-Core, 4) muuten default (invalid).
    /// </summary>
    private static Scene ResolveScene(Transform source, string sceneName)
    {
        // 1) Nimen perusteella
        if (!string.IsNullOrWhiteSpace(sceneName))
        {
            var byName = SceneManager.GetSceneByName(sceneName);
            if (byName.IsValid() && byName.isLoaded) return byName;
        }

        // 2) Lähdeobjektin scene (esim. kuollut unit, ase tms.)
        if (source != null)
        {
            var fromSource = source.gameObject.scene;
            if (fromSource.IsValid() && fromSource.isLoaded) return fromSource;
        }

        // 3) Ensimmäinen ladattu ei-Core-scene (nykyinen level)
```

```
        for (int i = 0; i < SceneManager.sceneCount; i++)
        {
            var s = SceneManager.GetSceneAt(i);
            if (s.isLoaded && !string.Equals(s.name, CoreSceneName, StringComparison.OrdinalIgnoreCase))
                return s;
        }

        // 4) Ei löytynyt – palautetaan invalid (kutsuja voi itse päättää mitä tekee)
        return default;
    }
}
```

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 OnDamaged;
    public event EventHandler OnDying;
    public event EventHandler OnDead;

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

    private bool isDying;
    private bool isDead;
    private Vector3 lastHitPosition;
    public Vector3 LastHitPosition => lastHitPosition;

    private int overkill;
    public int Overkill => overkill;

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

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

        health -= damageAmount;
        OnDamaged?.Invoke(this, EventArgs.Empty);

        if (health <= 0)
        {
            lastHitPosition = hitPosition;
            overkill = Math.Abs(health) + 1;
            health = 0;
            BeginDying();
        }
    }

    private void BeginDying()
    {
        if (isDying) return;
        isDying = true;

        var unit = GetComponent<Unit>();
        if (unit != null)
```

```
        {
            var allActions = unit.GetComponents<BaseAction>();
            foreach (var action in allActions)
            {
                if (action != null && action.IsActionActive())
                {
                    Debug.Log($"[HealthSystem] Pakkolopetetaan {action.GetType().Name} kuolevan unitin {unit.name} actionista");
                    action.ForceComplete();
                }
            }
        }

        UnitActionSystem.Instance?.UnlockInput();

        OnDying?.Invoke(this, EventArgs.Empty);
        StartCoroutine(FinalizeDeathNextFrame());
    }

    private System.Collections.IEnumerator FinalizeDeathNextFrame()
    {
        yield return null;
        FinalizeDeath();
    }

    public void FinalizeDeath()
    {
        if (isDead) return;
        isDead = true;
        OnDead?.Invoke(this, EventArgs.Empty);
    }

    public bool IsDying() => isDying;
    public bool IsDead() => isDead;

    public float GetHealthNormalized()
    {
        return (float)health / healthMax;
    }

    public int GetHealth()
    {
        return health;
    }

    public int GetHealthMax()
    {
        return healthMax;
    }

    public void ApplyNetworkHealth(int current, int max)
    {
        if (isDying || isDead) return;
```

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

Assets/scripts/Units/TeamsID.cs

```
using UnityEngine;

public class TeamsID : MonoBehaviour
{
    public static int CurrentTurnTeamId()
    {
        if (GameModeManager.SelectedMode == GameMode.Versus)
        {
            // Muunna PVP:n currentOwnerNetId -> teamId
            uint ownerNetId = PvPTurnCoordinator.Instance.currentOwnerNetId;
            return NetworkSync.IsOwnerHost(ownerNetId) ? 0 : 1; // host=0, client=1
        }
        // SP/Coop: TurnSystem pitää tiimin valmiina
        return (int)TurnSystem.Instance.CurrentTeam;
    }

}
```

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>
public enum Team { Player, Enemy }
[RequireComponent(typeof(HealthSystem))]
[RequireComponent(typeof(MoveAction))]
[RequireComponent(typeof(OverwatchAction))]
[RequireComponent(typeof(CoverSkill))]
public class Unit : NetworkBehaviour
{
    public Team Team;

    [SerializeField] public CoverSkill Cover { get; private set; }

    //  This is off long as this is under dvelopment...
    //  private const int ACTION_POINTS_MAX = 2;
    //  private int actionPoints = ACTION_POINTS_MAX;

    [SerializeField, Min(0)] private int ACTION_POINTS_MAX = 2;
    private int actionPoints;

    private int reactionPoins = 0;

    [SyncVar] public uint OwnerId;

    [SyncVar] private bool underFire = false;
    public bool IsUnderFire => underFire;

    public event Action<int, int> OnCoverPoolChanged;

    [SerializeField] public UnitArchetype archetype;
    [SerializeField] private WeaponDefinition currentWeapon;

    //Events
    public static event EventHandler OnAnyActionPointsChanged;
    public static event EventHandler ActionPointUsed;
    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 maxMoveDistance;

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

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

    private int grenadePCS;

    private bool _deathHandled;


    private void Awake()
    {
        renderers = GetComponentsInChildren<Renderer>(true);
        colliders = GetComponentsInChildren<Collider>(true);
        TryGetComponent(out anim);

        healthSystem = GetComponent<HealthSystem>();
        baseActionsArray = GetComponents<BaseAction>();
        maxMoveDistance = GetComponent<MoveAction>().GetMaxMoveDistance();
        Cover = GetComponent<CoverSkill>();
    }

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

        actionPoints = ACTION_POINTS_MAX;

        TurnSystem.Instance.OnTurnStarted += OnTurnStarted_HandleTurnStarted;
        TurnSystem.Instance.OnTurnEnded += OnTurnEnded_HandleTurnEnded;
        healthSystem.OnDead += HealthSystem_OnDead;

        OnAnyUnitSpawned?.Invoke(this, EventArgs.Empty);
        underFire = false;

        //****** Items ******
        grenadePCS = archetype.grenadeCapacity;

    }
```

```
    private void OnEnable()
    {
        if (Cover != null) Cover.OnCoverPoolChanged += ForwardCoverChanged;
        var hs = GetComponent<HealthSystem>();
        if (hs != null) hs.OnDying += HandleDying_ServerFirst;
    }

    private void OnDisable()
    {
        TurnSystem.Instance.OnTurnStarted -= OnTurnStarted_HandleTurnStarted;
        TurnSystem.Instance.OnTurnEnded -= OnTurnEnded_HandleTurnEnded;

        if (Cover != null) Cover.OnCoverPoolChanged -= ForwardCoverChanged;

        var hs = GetComponent<HealthSystem>();
        if (hs != null) hs.OnDying -= HandleDying_ServerFirst;
    }

    private void ForwardCoverChanged(int cur, int max) => OnCoverPoolChanged?.Invoke(cur, max);

    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;
    }


    [Server]
    public static void RaiseActionPointUsed(Unit unit)
    {
        ActionPointUsed?.Invoke(unit, EventArgs.Empty);
    }

    private void SpendActionPoints(int amount)
    {
        actionPoints -= amount;
        var team = Team;
       // Debug.Log("[Unit] Tiimi: " + Team + "Unitilla actionpisteitä nyt: " + actionPoints);

        // Nosta eventti vain authoritative-puolella:
```

```
        if (NetMode.ServerOrOff)
        {
            ActionPointUsed?.Invoke(this, EventArgs.Empty);
            OnAnyActionPointsChanged?.Invoke(this, EventArgs.Empty);
        }

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

        NetworkSync.BroadcastActionPoints(this, actionPoints);
    }



    public int GetActionPoints()
    {
        return actionPoints;
    }

    public int GetReactionPoints()
    {
      //  if (reactionPoins <= 0) { Debug.Log("[Unit] Tiimi: " + Team + "Unit ei voi ragoida. Reaktiopisteitä: " + reactionPoins); }
        return reactionPoins;
    }

    // Käytetään tilanteessa jossa toimitaan toisen vuorolla, kuten Overwach.
    public void SpendReactionPoints()
    {
        if(reactionPoins > 0)
        {
            reactionPoins -= 1;
        }
    }

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

    public bool IsEnemy()
    {
        return isEnemy;
    }

    public bool IsDying()
    {
        return healthSystem != null && healthSystem.IsDying();
```

```
    }

    public bool IsDead()
    {
        return healthSystem != null && healthSystem.IsDead();
    }

    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(1.30f));
    }

    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;
    }

    public void SetUnderFire(bool value)
    {
        if (IsDying() || IsDead()) return;

        if (!NetworkServer.active && !NetworkClient.active)
        {
            underFire = value;
            return;
        }

        if (NetworkServer.active)
        {
            SetUnderFireServer(value);
            return;
        }

        var ni = GetComponent<NetworkIdentity>();
        if (NetworkClient.active && NetworkSyncAgent.Local != null && ni != null)
        {
            if (this == null || IsDying() || IsDead())
            {
                return;
            }

            NetworkSyncAgent.Local.CmdSetUnderFire(ni.netId, value);
        }
    }

    [Server]
    public void SetUnderFireServer(bool value)
    {
        if (IsDying() || IsDead()) return;

        underFire = value;
        var agent = FindFirstObjectByType<NetworkSyncAgent>();
        if (agent != null)
            agent.ServerBroadcastUnderFire(this, value);
    }

    public void ApplyNetworkUnderFire(bool value)
    {
        underFire = value;
    }

    // ****** Cover Skill ***********
    public int GetPersonalCover() => Cover ? Cover.GetPersonalCover() : 0;
```

```
    public int GetPersonalCoverMax() => Cover ? Cover.GetPersonalCoverMax() : 1;
    public float GetCoverNormalized() => Cover ? Cover.GetCoverNormalized() : 0f;
    public int GetCoverRegenPerUnusedAP() => Cover ? Cover.GetCoverRegenPerUnusedAP() : 0;

    public bool HasMoved() => Cover ? Cover.HasMoved() : false;

    // public void SetPersonalCover(int v) { if (Cover) Cover.SetPersonalCover(v); }
    public void SetPersonalCover(int v)
    {
        if (IsDying() || IsDead()) return;
        if (Cover) Cover.SetPersonalCover(v);
    }

    //public void SetCoverBonus() { if (Cover) Cover.ServerApplyCoverBonus(); }
    public void SetCoverBonus()
    {
        if (IsDying() || IsDead()) return;
        if (Cover) Cover.ServerApplyCoverBonus();
    }


    public int GetCurrentCoverBonus() => Cover ? Cover.GetCurrentCoverBonus() : 0;
    public void ResetCurrentCoverBonus() { if (Cover) Cover.ResetCurrentCoverBonus(); }

    public void RegenCoverBy(int amount) { if (Cover) Cover.RegenCoverBy(amount); }
    public void RegenCoverOnMove(int distance) { if (Cover) Cover.RegenCoverOnMove(distance); }

    public void ApplyNetworkCover(int cur, int max) { if (Cover) Cover.ApplyNetworkCover(cur, max); }

    //*********************************

    // ***** weapons ******
    public void UseGrenade()
    {
        if (grenadePCS <= 0)
        {
            grenadePCS = 0;
            return;
        }
        grenadePCS -= 1;
    }

    public int GetGrenadePCS() => grenadePCS;

    public WeaponDefinition GetCurrentWeapon()
    {
        return currentWeapon;
    }

    private int _lastApStartTurnId = -1;
    private void OnTurnStarted_HandleTurnStarted(Team startTurnTeam, int turnId)
    {
```

```
        if (NetworkClient.active && !NetworkServer.active) return;
        if (Team != startTurnTeam) return;            // vain oman puolen alussa
        if (_lastApStartTurnId == turnId) return;       // duplikaattisuojaksi
        _lastApStartTurnId = turnId;

        actionPoints = ACTION_POINTS_MAX;
        reactionPoins = 0;
        OnAnyActionPointsChanged?.Invoke(this, EventArgs.Empty);

        // LISÄÄ TÄMÄ: Broadcastaa AP-muutos myös verkossa
        if (NetworkServer.active || NetworkClient.active)
        {
            NetworkSync.BroadcastActionPoints(this, actionPoints);
        }
    }

    private void OnTurnEnded_HandleTurnEnded(Team endTurnTeam, int turnId)
    {
        if (NetworkClient.active && !NetworkServer.active) return;
        if (Team != endTurnTeam) return;            // vain sen puolen lopussa joka oli vuorossa
        int ap = GetActionPoints();
        int per = GetCoverRegenPerUnusedAP();           // palauttaa >0 vain jos ei underFire
        if (ap > 0 && per > 0) Cover.RegenCoverBy(ap * per);  // coverSkill hoitaa clampit jne.


        reactionPoins = actionPoints;

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

    public int GetTeamID()
    {
        if (NetMode.Offline) // !NetworkServer.active && !NetworkClient.active
        {
            // Offline: käytä isEnemy flagia
            return isEnemy ? 1 : 0;
        }

        // Online: Versus vs Co-op
        var mode = GameModeManager.SelectedMode;
        if (mode == GameMode.Versus)
        {
            // CLIENTILLA: tarkista onko tämä unitti omistettu täällä
            if (NetworkClient.active && !NetworkServer.active)
            {
                var ni = GetComponent<NetworkIdentity>();
                if (ni != null)
                {
                    // Jos tämä on mun unitti, palauta 1 (Client team)
                    // Muuten palauta 0 (Host team)
                    return ni.isOwned ? 1 : 0;
                }
```

```
            }

            // SERVERILLÄ/HOSTILLA: käytä OwnerId-tarkistusta
            return NetworkSync.IsOwnerHost(OwnerId) ? 0 : 1;
        }

        // Co-Op / SinglePlayer: pelaajat = 0, viholliset = 1
        return isEnemy ? 1 : 0;
    }

    private void FreezeBeforeDespawn()
    {
        // 1) Estä AE/NetworkAnimator-lähetykset varmasti
        if (TryGetComponent<UnitAnimator>(out var ua))
        {
            var na = ua.GetComponent<NetworkAnimator>();
            if (na != null)
            {
                na.enabled = false;
            }
        }

        // 2) Estä aseiden näkyvyys-Commandit (hae lapsista)
        var vis = GetComponentInChildren<WeaponVisibilitySync>(true); // true jos haluat löytää myös inaktiivit
        if (vis != null)
        {
            vis.enabled = false;
        }

        // 3) Estä myöhästyneet action-päivitykset
        foreach (var ba in GetComponents<BaseAction>())
        {
            ba.enabled = false;
        }

        var lv = GetComponent<LocalVisibility>();
        if (lv == null) lv = gameObject.AddComponent<LocalVisibility>();
        lv.Apply(false);  // disabloi kaikki renderöijät ja canvasit hierarkiasta

        // Piilota maailman-UI heti
        var worldUi = GetComponentInChildren<UnitWorldUI>(true);
        if (worldUi != null) worldUi.SetVisible(false);

        // --- UUTTA: viimeinen varmistus että input/UI vapautuu ---
        UnitActionSystem.Instance.UnlockInput();

    }

    // <<< Kuolemaketjun alku (server) >>>
    /*
    [Server]
    private void HandleDying_ServerFirst(object sender, System.EventArgs e)
```

```
    {
        if (!isServer) return;
        if (_deathHandled) return;
        _deathHandled = true;

        // a) Peilaa jäädytys clientille heti:
        RpcFreezeClientSide();

        // b) Katkaise serverin liike deterministisesti + snap (tyhjentää clientin interp-puskurit)
        DeathStopper.TryHalt(this);

        // c) Sammuta serverillä (nykyinen teidän metodi)
        FreezeBeforeDespawn();
    }
    */

    private void HandleDying_ServerFirst(object sender, System.EventArgs e)
    {
        if (_deathHandled) return;
        _deathHandled = true;

        if (NetworkServer.active)
        {
            // online/server-polku
            RpcFreezeClientSide();          // ajetaan vain jos server aktiivinen
            DeathStopper.TryHalt(this);     // server-varmistus liikkeen pysäytykseen
            FreezeBeforeDespawn();          // teidän nykyinen sulku ennen despawnia
        }
        else
        {
            // offline-polku: ei RPC:tä
            DeathStopper.TryHaltOfline(this);
            FreezeBeforeDespawn();
        }
    }

    [ClientRpc]
    void RpcFreezeClientSide()
    {
        // 1) Katkaise clientin actionit
        foreach (var ba in GetComponents<BaseAction>()) ba.enabled = false;

        // 2) Sammuta animaatio- ja ase-synkat
        var ua = GetComponent<UnitAnimator>();
        if (ua)
        {
            var na = ua.GetComponent<NetworkAnimator>();
            if (na) na.enabled = false;
        }
        var anim = GetComponentInChildren<Animator>(true);
        if (anim) anim.enabled = false;
```

```
        var vis = GetComponentInChildren<WeaponVisibilitySync>(true);
        if (vis) vis.enabled = false;

        // 3) Piilota KAIKKI renderöijät+canvasit → myös valintarengas
        var lv = GetComponent<LocalVisibility>();
        if (lv == null) lv = gameObject.AddComponent<LocalVisibility>();
        lv.Apply(false);

        // 4) Piilota world-space UI varmuudeksi
        var worldUI = GetComponentInChildren<UnitWorldUI>(true);
        if (worldUI) worldUI.SetVisible(false);

        // 5) Vapauta UI (käytä teidän olemassa olevaa metodia)
        UnitActionSystem.Instance.UnlockInput();

        // -- VAPAUTA KAMERA / INPUT AINA --
        CameraThaw.Thaw("Unit dying (RpcFreezeClientSide)");
    }
}
```

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;

    //private HealthSystem ownerHealth;

    protected virtual void Awake()
    {
        /*
        if (ownerHealth == null)
            ownerHealth = GetComponentInParent<HealthSystem>();
        */

        unit = GetComponent<Unit>();
    }

    void OnEnable()
    {
        /*
        if (ownerHealth == null) ownerHealth = GetComponentInParent<HealthSystem>();
        if (ownerHealth != null)
        {
            ownerHealth.OnDying += HandleUnitDying;
            // DEBUG
            Debug.Log($"[BaseAction:{GetType().Name}] Subscribed OnDying for {unit?.name}");
        }
        else
        {
            Debug.LogWarning($"[BaseAction:{GetType().Name}] ownerHealth == null on {gameObject.name}");
        }
        */
    }

    void OnDisable()
```

```
    {
        /*
        if (ownerHealth != null)
            ownerHealth.OnDying -= HandleUnitDying;
        */

    }

    /*
    private void HandleUnitDying(object sender, EventArgs e)
    {
        ForceCompleteNow();
    }

    // DODO Testaa toimiiko tämä AI.n jumiutumisessa.

    protected virtual void ForceCompleteNow()
    {

        if (!isActive) return;
        isActive = false;
        Debug.Log("[BaseAction] Set isActive: " + isActive);
        Action callback = onActionComplete;
        Debug.Log("[BaseAction] onActionComplete: " + onActionComplete);
        Debug.Log("[BaseAction] Action callback: " + callback);
        onActionComplete = null;
        Debug.Log("[BaseAction] onActionComplete: " + onActionComplete);
        StopAllCoroutines();
        Debug.Log("[BaseAction] StopAllCorroutines");

        OnAnyActionCompleted?.Invoke(this, EventArgs.Empty);
    }
    */
    public bool IsActionActive()
    {
        return isActive;
    }

    public static bool AnyActionActive()
    {
        var allActions = FindObjectsByType<BaseAction>(FindObjectsSortMode.None);
        foreach (var action in allActions)
        {
            if (action != null && action.isActive)
            {
                return true;
            }
        }
        return false;
    }

    public static void ForceCompleteAllActiveActions()
```

```
    {
        var allActions = FindObjectsByType<BaseAction>(FindObjectsSortMode.None);
        int forcedCount = 0;

        foreach (var action in allActions)
        {
            if (action != null && action.isActive)
            {
                Debug.LogWarning($"[BaseAction] Pakkolopetetaan aktiivinen action: {action.GetType().Name} unitilta {action.unit?.name}");
                action.ForceComplete();
                forcedCount++;
            }
        }

        if (forcedCount > 0)
        {
            Debug.LogWarning($"[BaseAction] Pakkolopetettiin {forcedCount} actionia");
        }
    }

    public void ForceComplete()
    {
        if (!isActive) return;

        isActive = false;

        Action callback = onActionComplete;
        onActionComplete = null;

        StopAllCoroutines();

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

        callback?.Invoke();
    }

    // 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)
    {
        // Tarkoituksia perutaan vain pelaajan vuorolla. (Tai vuoron alussa)
        // jotkin actionit jatkuvat vuorojen yli.
        if(TurnSystem.Instance.IsPlayerTurn())
        {
            CanselAllIntents();
        }

        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()
    {
        if (!isActive) { return; }

        isActive = false;
        onActionComplete();
        OnAnyActionCompleted?.Invoke(this, EventArgs.Empty);
    }

    public void ResetChostActions()
    {
        ActionComplete();
    }

    // Perutaan kaikki Unitin aikomukset jos Unit tekee jotakin muuta.
    private void CanselAllIntents()
    {
        unit.GetComponent<OverwatchAction>().CancelOverwatchIntent();
    }

    public Unit GetUnit()
    {
        return unit;
    }

    public void MakeDamage(int damage, Unit targetUnit)
    {
        if (targetUnit == null || targetUnit.IsDying() || targetUnit.IsDead()) return;
```

```
        Vector3 attacerPos = unit.GetWorldPosition() + Vector3.up * 1.6f;
        Vector3 targetPos = targetUnit.GetWorldPosition() + Vector3.up * 1.2f;
        Vector3 dir = targetPos - attacerPos;

        if (dir.sqrMagnitude < 0.0001f) dir = targetUnit.transform.forward;
        dir.Normalize();

        float backOffset = 0.7f;
        Vector3 hitPosition = targetPos - dir * backOffset;

        Vector3 side = Vector3.Cross(dir, Vector3.up).normalized;
        hitPosition += side * UnityEngine.Random.Range(-0.1f, 0.1f);

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

    public void ApplyHit(int damage, bool bypassCover, bool coverOnly, Unit targetUnit, bool melee = false)
    {
        if (targetUnit == null || targetUnit.IsDying() || targetUnit.IsDead()) return;

        targetUnit.SetUnderFire(true);
        var ct = GetCoverType(targetUnit);

        if (ct == CoverService.CoverType.None && !melee)
        {
            MakeDamage(damage, targetUnit);
            return;
        }

        float mitigate = 1;
        if (targetUnit.GetPersonalCover() > 0)
        {
            mitigate = CoverService.GetCoverMitigationPoints(ct);
        }

        int toCover = Mathf.RoundToInt(damage * mitigate);
        int before = targetUnit.GetPersonalCover();
        int after = before - toCover;

        if (melee)
        {
            after -= damage;
        }

        if (after >= 0)
        {
            targetUnit.SetPersonalCover(after);
            NetworkSync.UpdateCoverUI(targetUnit);

            // Vaikka suojaa jäisi niin kriittinen osuma tekee vahinkoa silti.
            if(bypassCover) MakeDamage(damage/3, targetUnit);
```

```
        }
        else
        {
            targetUnit.SetPersonalCover(0);
            NetworkSync.UpdateCoverUI(targetUnit);

            // Ainoastaan oikeat osumat voivat tehdä vahinkoa.
            if (!coverOnly && !bypassCover) MakeDamage(damage - before, targetUnit);

            // Kriittinen osuma osuu kokonaisuudessaan.
            if(bypassCover) MakeDamage(damage, targetUnit);
        }
    }

    public CoverService.CoverType GetCoverType(Unit targetUnit)
    {
        var gp = targetUnit.GetGridPosition();
        var node = PathFinding.Instance.GetNode(gp.x, gp.z, gp.floor);
        var ct = CoverService.EvaluateCoverHalfPlane(unit.GetGridPosition(), targetUnit.GetGridPosition(), node);
        return ct;
    }


    public bool RotateTowards(Vector3 targetPosition, float rotSpeedDegPerSec = 720f, float epsilonDeg = 2f)
    {
        Vector3 to = targetPosition - transform.position;
        to.y = 0f;

        if (to.sqrMagnitude < 1e-6f) return true;

        Vector3 dir = to.normalized;
        Quaternion desired = Quaternion.LookRotation(dir, Vector3.up);

        transform.rotation = Quaternion.RotateTowards(
            transform.rotation, desired, rotSpeedDegPerSec * Time.deltaTime
        );

        return Quaternion.Angle(transform.rotation, desired) <= epsilonDeg;
    }

    /*
    public bool RotateTowards(Vector3 worldTarget, float speedDegPerSec)
    {
        Vector3 to = worldTarget - transform.position; to.y = 0f;
        if (to.sqrMagnitude < 1e-6f) return true;

        Quaternion desired = Quaternion.LookRotation(to.normalized, Vector3.up);
        float step = speedDegPerSec * Time.deltaTime;               // <-- avain
        transform.rotation = Quaternion.RotateTowards(transform.rotation, desired, step);
        return Quaternion.Angle(transform.rotation, desired) < 0.5f;
    }
*/
```

```
    // -------------- 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
{
    [Header("Preview")]
    [SerializeField] private GrenadeArcPreview arcPreview; // LineRenderer + GrenadeArcPreview
    [SerializeField] private float cellSizeWU = 2f;        // esim. LevelGrid.Instance.CellSize
    [SerializeField] private Transform throwPoint;         // esim. UnitAnimator.rightHandTransform (optional)
    [SerializeField] private ThrowArcConfig throwArcConfig; // sama assetti kuin projektiililla ja preview'lla
    [SerializeField] private LayerMask arcBlockMask;        // esim. "Environment" tms. (EI Units)
    [SerializeField] private int minSegments = 12;
    [SerializeField] private int maxSegments = 36;


    // --- LASKEUTUMISEN SÄÄDÖT ---
    [SerializeField] private LayerMask landingObstacleMask;   // esim. Obstacles (ruudussa olevat laatikot yms.)
    [SerializeField] private float maxLandingObstacleHeight = 0.35f; // kuinka korkea "matala este" voi olla (WU)
    [SerializeField] private float landingBoxShrink = 0.49f;  // 0.45–0.49: ettei nappaa naapuriruutua
    [SerializeField] private float landingBoxHalfY = 0.75f;   // ruudun yläpuolelle ulottuva haku (WU)

    // --- ARC SÄÄDÖT (jos et jo määrittänyt) ---
    [SerializeField] private float grenadeRadius = 0.12f;     // kranaatin "paksuus" törmäystesteille
    [SerializeField] private float arcLift = 0.2f;            // pieni nosto, ettei suutele lattiaa
    [SerializeField] private float heightClearance = 0.05f;   // toleranssi esteen yläreunaan


    // (Prefabi-viite saa jäädä, mutta tätä tiedostoa ei käytetä spawniin)
    [SerializeField] private Transform grenadeProjectilePrefab;

    [SerializeField] private LayerMask mousePlaneMask;   // vain MousePlane-kerrokset
    [SerializeField] private float mousePlaneHalfThickness = 0.05f; // pystysuunnan puolipaksuus
    [SerializeField] private float mousePlaneYOffset = 0.01f;      // pieni nosto keskelle laatikkoa

    [SerializeField] private LayerMask ceilingMask;

    public event EventHandler ThrowGranade;
    public event EventHandler ThrowReady;

    public Vector3 TargetWorld { get; private set; }

    private GridPosition? _lastHover;
   // private bool _wasActive;  // reunan tunnistus
    private bool _wasSelected;

    protected override void Awake()
    {
        base.Awake();
```

```
        // Jos throwPointia ei ole asetettu inspectorissa, yritetään hakea UnitAnimatorilta
        if (throwPoint == null)
        {
            var anim = GetComponentInChildren<UnitAnimator>(true);
            if (anim != null && anim.GetrightHandTransform() != null)
                throwPoint = anim.GetrightHandTransform();
        }

        // Alusta preview'n perusasetukset kerran
        if (arcPreview != null)
        {
            arcPreview.SetOrigin(throwPoint != null ? throwPoint : transform);
            arcPreview.SetCellSize(cellSizeWU);
        }
    }
    void OnEnable()  => GrenadeProjectile.OnAnyGranadeExploded += OnGrenadeEnded;



    private void OnDisable()
    {
        GrenadeProjectile.OnAnyGranadeExploded -= OnGrenadeEnded;
        if(arcPreview != null)
        {
            arcPreview.Hide();
        }
        _lastHover = null;
     //   _wasActive = false;
    }

    private void Update()
    {
        // 0) Onko tämä action tällä hetkellä valittuna?
        bool isSelected =
            UnitActionSystem.Instance != null &&
            UnitActionSystem.Instance.GetSelectedAction() == this;

        // 1) Reunan tunnistus valituksi tulossa/poistumassa
        if (isSelected && !_wasSelected)
        {
            // hae Core-scenessä oleva preview, jos viite puuttuu
            if (arcPreview == null)
                arcPreview = FindFirstObjectByType<GrenadeArcPreview>(FindObjectsInactive.Exclude);

            // hae heittopiste, jos puuttuu
            if (throwPoint == null)
            {
                var anim = GetComponentInChildren<UnitAnimator>(true);
                if (anim != null && anim.GetrightHandTransform() != null)
                    throwPoint = anim.GetrightHandTransform();
            }
```

```
            if (arcPreview != null)
            {
                arcPreview.SetOrigin(throwPoint != null ? throwPoint : transform);
                arcPreview.SetCellSize(cellSizeWU);
            }

            _lastHover = null; // pakota eka päivitys
        }
        else if (!isSelected && _wasSelected)
        {
            arcPreview?.Hide();
            _lastHover = null;
        }
        _wasSelected = isSelected;

        if (!isSelected) return;

        // 2) Päivitä viiva vain, kun hover-ruutu vaihtuu ja ruutu on validi
        GridPosition gp = LevelGrid.Instance.GetGridPosition(MouseWorld.GetMouseWorldPosition());
        if (!LevelGrid.Instance.IsValidGridPosition(gp))
        {
            arcPreview?.Hide();
            _lastHover = null;
            return;
        }

        if (_lastHover.HasValue && _lastHover.Value == gp) return;
        _lastHover = gp;

        // 3) Piirrä kaari
        Vector3 targetW = LevelGrid.Instance.GetWorldPosition(gp);
        arcPreview?.ShowArcTo(targetW);
    }

    public override string GetActionName() => "Granade";

    private void OnGrenadeEnded(object sender, EventArgs e)
    {
        GetValidGridPositionList();            // teillä jo oleva metodi
        GridSystemVisual.Instance?.UpdateGridVisuals();
    }

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


    public override List<GridPosition> GetValidGridPositionList()
    {
        unit ??= GetComponent<Unit>();
        var result = new List<GridPosition>();
        if (unit == null || LevelGrid.Instance == null) return result;
```

```
        int floorsUp = 2;      // montako ylös skannataan
        int floorsDown = 2;    // montako alas (0 jos et halua)

        // Heittopiste
        Transform originT = throwPoint;
        if (originT == null)
        {
            var anim = GetComponentInChildren<UnitAnimator>(true);
            if (anim != null && anim.GetrightHandTransform() != null)
                originT = anim.GetrightHandTransform();
        }
        if (originT == null) originT = transform;
        Vector3 origin = originT.position;

        GridPosition unitGP = unit.GetGridPosition();
        int rangeTiles = unit.archetype.throwingRange;

        var topmostByXZ = new Dictionary<string, GridPosition>();

        for (int dx = -rangeTiles; dx <= rangeTiles; dx++)
        for (int dz = -rangeTiles; dz <= rangeTiles; dz++)
        {
            int cost = SircleCalculator.Sircle(dx, dz);
            if (cost > 10 * rangeTiles) continue;

            for (int df = -floorsDown; df <= floorsUp; df++)
            {
                var test = new GridPosition(unitGP.x + dx, unitGP.z + dz, unitGP.floor + df);
                if (!LevelGrid.Instance.IsValidGridPosition(test)) continue;

                // 1) VAATIMUS: ruudussa täytyy olla MousePlane ko. kerroksessa
                if (!HasMousePlaneAt(test)) continue;

                // 2) kohderuudun laskeutuminen ok?
                if (!CheckDestinationClearance(test, origin)) continue;

                // 2) ARC-CHECK vasta sen jälkeen (säästää fysiikkaa)
                Vector3 targetW = LevelGrid.Instance.GetWorldPosition(test);

                int segs;
                    if (throwArcConfig != null)
                    {
                        float d = Vector2.Distance(new Vector2(origin.x, origin.z), new Vector2(targetW.x, targetW.z));
                        segs = Mathf.Clamp(throwArcConfig.EvaluateSegments(d, Mathf.Max(0.01f, cellSizeWU)), minSegments, maxSegments);
                    }
                    else segs = minSegments;

                float apexClamped = ArcApexSolver.ComputeCeilingClampedApex(
                    origin, targetW, throwArcConfig, ceilingMask,
                    ceilingClearance: 0.08f, samples: segs);

                bool clear = ArcVisibility.IsArcClear(
```

```
                    start: origin,
                    end: targetW,
                    cfg: throwArcConfig,
                    segments: segs,
                    mask: arcBlockMask,
                    ignoreRoot: (throwPoint ? throwPoint : transform).root,
                    lift: 0.2f,
                    capsuleRadius: throwArcConfig.GetCapsuleRadius(cellSizeWU),
                    heightClearance: 0.05f,
                    tStart: 0.02f,
                    tEnd: 0.95f,
                    cellSizeWU: cellSizeWU,
                    fullTilePerc: 0.8f,
                    tallWallY: 0.6f,
                    apexOverrideWU: apexClamped
                );
                    if (!clear) continue;


                string key = $"{test.x},{test.z},{test.floor}";
                if (!topmostByXZ.ContainsKey(key))
                    topmostByXZ[key] = test;

            }
        }

        foreach (var kv in topmostByXZ) result.Add(kv.Value);
        return result;
    }

    private bool HasMousePlaneAt(GridPosition gp)
    {
        Vector3 center = LevelGrid.Instance.GetWorldPosition(gp);
        // laatikko hieman pienempi kuin laatta, ettei “nuolaise” naapuria
        Vector3 halfExtents = new Vector3(cellSizeWU * 0.49f, mousePlaneHalfThickness, cellSizeWU * 0.49f);
        center.y += mousePlaneYOffset; // varmuus: korkeudessa pieni toleranssi

        // Collide: lasketaan myös trigger-MousePlane't
        return Physics.CheckBox(center, halfExtents, Quaternion.identity, mousePlaneMask, QueryTriggerInteraction.Collide);
    }

    private bool CheckDestinationClearance(GridPosition gp, Vector3 origin)
    {
        Vector3 center = LevelGrid.Instance.GetWorldPosition(gp);
        float floorY = center.y;

        Vector3 half = new Vector3(cellSizeWU * landingBoxShrink, landingBoxHalfY, cellSizeWU * landingBoxShrink);
        var cols = Physics.OverlapBox(center + Vector3.up * (landingBoxHalfY * 0.5f),
                                      half,
                                      Quaternion.identity,
                                      landingObstacleMask,
                                      QueryTriggerInteraction.Collide);
```

```
        if (cols == null || cols.Length == 0)
            return true; // tyhjä ruutu → ok

        float maxTopRel = 0f;
        bool hasAllowingOverride = false;

        foreach (var c in cols)
        {
            if (!c) continue;
            float topRel = c.bounds.max.y - floorY;
            if (topRel > maxTopRel) maxTopRel = topRel;
        }

        // Perussääntö korkeudelle
        bool heightOK = (maxTopRel <= maxLandingObstacleHeight) || hasAllowingOverride;
        if (!heightOK) return false;

        // BONUS: jos ruudussa oli JOTAIN (maxTopRel > 0), tarkista myös laskeutumiskulma
        if (maxTopRel > 0.01f && throwArcConfig != null)
        {
            float angleDeg = ArcMath.ComputeDescentAngleDeg(
                origin, center, throwArcConfig, lift: arcLift);

            if (angleDeg < throwArcConfig.minLandingAngleDegForObstacles)
                return false; // liian loiva → ei laskeuduta esteen päälle
        }

        return true;
    }

    public float GetMaxThrowRangeWU()
    {
        // Yritä käyttää ThrowArcConfigin farRangeWU:ta jos se on asetettu
        if (throwArcConfig != null && throwArcConfig.farRangeWU > 0f)
            return throwArcConfig.farRangeWU;

        // Muuten: tiles * cellsize (oma cellSizeWU toimii varana jos LevelGrid null)
        float cs = (LevelGrid.Instance != null) ? LevelGrid.Instance.GetCellSize() : cellSizeWU;
        var u = unit != null ? unit : GetComponent<Unit>();
        int tiles = (u != null && u.archetype != null) ? u.archetype.throwingRange : 6;
        return tiles * cs;
    }


    // 2) Kutsukääre radan tarkastukselle (käyttää jo teillä olevaa ArcVisibilityä)
    private bool CheckArcClear(Vector3 start, Vector3 end, int segs)
    {
        // Huom: leikataan loppukärkeä hieman (tEnd=0.95), jotta laskeutumisruudun esineet
        // eivät blokkaa rataa – niiden arvio tekee CheckDestinationClearance erikseen.
        return ArcVisibility.IsArcClear(
            start: start,
```

```
            end: end,
            cfg: throwArcConfig,
            segments: segs,
            mask: arcBlockMask,                 // Walls/Ceilings/Thick obstacles (EI Units)
            ignoreRoot: transform,
            lift: arcLift,
            capsuleRadius: grenadeRadius,
            heightClearance: heightClearance,
            tStart: 0.02f,
            tEnd: 0.95f
        );
    }

    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        GetUnit().UseGrenade();
        ActionStart(onActionComplete);

        TargetWorld = LevelGrid.Instance.GetWorldPosition(gridPosition);

        // Päätös tehty → piilota preview nyt (spawn ja animaatiot tapahtuvat muualla kuten ennen)
        if (arcPreview != null) arcPreview.Hide();

        StartCoroutine(TurnAndThrow(0.5f, TargetWorld));
    }

    public override int GetActionPointsCost()
    {
        return 2;
    }

    private IEnumerator TurnAndThrow(float delay, Vector3 targetWorld)
    {
        float waitAfterAligned = 0.1f;
        float alignedTime = 0f;

        // "Deadman Switch" Varmistaa heiton tietyn ajan kuluttua vaikka kääntyminen kohti kohdetta ei olisi täydellinen.
        float maxAlignTime = 1.5f;
        float elapsed = 0f;

        while (true)
        {
            bool aligned = RotateTowards(targetWorld, 750);
            if (aligned)
            {
                alignedTime += Time.deltaTime;
                if (alignedTime >= waitAfterAligned) break;
            }
            else
            {
                alignedTime = 0f;
```

```
                elapsed += Time.deltaTime;
                if (elapsed >= maxAlignTime)
                {
                    Debug.LogWarning("[Grenade] Align timeout → throwing anyway.");
                    break;
                }
            }
            yield return null;
        }

        // Täsmälleen sama eventti kuin ennen — UnitAnimator hoitaa näkyvyydet ja AE-triggerin
        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(), 750))
                    {
                        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;
                ApplyHit(damage, false, false, targetUnit, true);
                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()
    {
        var valid = new List<GridPosition>();
        GridPosition origin = unit.GetGridPosition();

        for (int dx = -maxMeleedDistance; dx <= maxMeleedDistance; dx++)
        {
            for (int dz = -maxMeleedDistance; dz <= maxMeleedDistance; dz++)
            {
                if (dx == 0 && dz == 0) continue; // ei itseään

                var gp = origin + new GridPosition(dx, dz, 0);

                // 1) RAJAT ENSIN -> estää out-of-range -virheen
                if (!LevelGrid.Instance.IsValidGridPosition(gp)) continue;

                // Manhattan -> sulkee diagonaalit
                // if (Mathf.Abs(dx) + Mathf.Abs(dz) > maxMeleedDistance) continue;

                // Chebyshev -> sallii diagonaalit
                if (Mathf.Max(Mathf.Abs(dx), Mathf.Abs(dz)) > maxMeleedDistance) continue;

                // 2) onko ruudussa ketään?
                if (!LevelGrid.Instance.HasAnyUnitOnGridPosition(gp)) continue;

                var target = LevelGrid.Instance.GetUnitAtGridPosition(gp);
                if (target == null || target == unit) continue;           // varmistus
                if (target.IsEnemy() == unit.IsEnemy()) continue;         // ei omia
```

```
                valid.Add(gp);
            }
        }
        return valid;
    }

    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 Mirror;
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;

    GridPosition thisTurnStartingGridPosition;
    GridPosition thisTurnEndridPosition;

    private GridPosition _lastVisionPos;

    [SerializeField] private int maxMoveDistance = 4;

    private int distance;

    private List<Vector3> positionList;
    private int currentPositionIndex;

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

    private void Start()
    {
        distance = 0;
        thisTurnStartingGridPosition = unit.GetGridPosition();
        thisTurnEndridPosition = unit.GetGridPosition();
        TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;

    }

    void OnDisable()
    {
        TurnSystem.Instance.OnTurnChanged -= TurnSystem_OnTurnChanged;

    }

    private void TurnSystem_OnTurnChanged(object sender, EventArgs e)
    {
        thisTurnStartingGridPosition = unit.GetGridPosition();
        distance = 0;
```

```
    }

    private void Update()
    {

        if (unit.IsDying() || unit.IsDead())
        {
            ForceStopNow();
            return;
        }

        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;

            float rotationSpeed = 10f;
            transform.forward = Vector3.Slerp(transform.forward, moveDirection, Time.deltaTime * rotationSpeed);

            float moveSpeed = 6f;
            transform.position += moveSpeed * Time.deltaTime * moveDirection;
        }

        float stoppingDistance = 0.2f;
        if (Vector3.Distance(transform.position, targetPosition) < stoppingDistance)
        {
            var levelGrid = LevelGrid.Instance;
            if (levelGrid != null)
            {
                var currentLevelGrid = levelGrid.GetGridPosition(unit.transform.position);

                if (!currentLevelGrid.Equals(_lastVisionPos))
                {
                    if (unit != null && !unit.IsDying() && !unit.IsDead())
```

```
                {
                    var unitVision = unit.GetComponent<UnitVision>();
                    if (unitVision != null && unitVision.IsInitialized)
                        unitVision.NotifyMoved();
                }

                // UUSI: Ilmoita serverille grid-muutoksesta
                if (NetworkServer.active || NetworkSync.IsOffline)
                {
                    // Server: tarkista overwatch suoraan
                    StatusCoordinator.Instance.CheckOverwatchStep(unit, currentLevelGrid);
                }
                else if (NetworkClient.active && NetworkSyncAgent.Local != null)
                {
                    // Client: pyydä serveriä tarkistamaan overwatch
                    var ni = unit.GetComponent<NetworkIdentity>();
                    if (ni != null)
                    {
                        NetworkSyncAgent.Local.CmdCheckOverwatchStep(ni.netId, currentLevelGrid.x, currentLevelGrid.z, currentLevelGrid.floor);
                    }
                }

                _lastVisionPos = currentLevelGrid;
            }
        }

        thisTurnEndridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
        DistanceFromStartingPoint();

        currentPositionIndex++;
        if (currentPositionIndex >= positionList.Count)
        {
            // Tarkista vielä kerran ennen network-kutsuja
            if (unit != null && !unit.IsDying() && !unit.IsDead())
            {
                if (thisTurnStartingGridPosition != thisTurnEndridPosition)
                {
                    unit.SetUnderFire(false);
                    CheckAndApplyCoverBonus();
                }
                else
                {
                    unit.SetUnderFire(true);
                    CheckAndApplyCoverBonus();
                }

                unit.GetComponent<UnitVision>().NotifyMoved();
            }

            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)
                {
                    isChangingFloors = true;
                    differentFloorsTeleportTimer = differentFloorsTeleportTimerMax;
                }
            }
        }
    }

    public void ForceStopNow()
    {
        StopAtCurrentPosition();
        positionList?.Clear();
        currentPositionIndex = 0;
        isChangingFloors = false;

        // Merkitse loppuruutu
        thisTurnEndridPosition = unit.GetGridPosition();

        // Lopeta action
        OnStopMoving?.Invoke(this, EventArgs.Empty);
        ActionComplete();
    }

    public void StopAtCurrentPosition()
    {

        GridPosition currentPos = unit.GetGridPosition();

        // Laske uusi polku nykyisestä ruudusta samaan ruutuun (= tyhjä polku)
        List<GridPosition> pathGridPositionsList = PathFinding.Instance.FindPath(
            currentPos,
            currentPos,
            out int pathLength,
            maxMoveDistance
        );

        // Päivitä position lista
        currentPositionIndex = 0;
        positionList = new List<Vector3>();

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

```
        // Pysäytä välittömästi
        if (positionList == null || positionList.Count <= 1)
        {
            // Ei polkua tai vain nykyinen ruutu -> lopeta liike
            thisTurnEndridPosition = currentPos;
            OnStopMoving?.Invoke(this, EventArgs.Empty);
            ActionComplete();
        }
    }


    private void CheckAndApplyCoverBonus()
    {
        if (unit == null || unit.IsDying() || unit.IsDead()) return;

        if (!NetworkServer.active && !NetworkClient.active)
        {
            unit.SetCoverBonus();
            return;
        }

        if (NetworkServer.active)
        {
            unit.SetCoverBonus();
            return;
        }

        if (NetworkClient.active && NetworkSyncAgent.Local != null)
        {
            var ni = unit.GetComponent<NetworkIdentity>();
            if (ni != null && ni.netId != 0)
            {
                NetworkSyncAgent.Local.CmdApplyCoverBonus(ni.netId);
            }
        }
    }


    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        _lastVisionPos = unit.GetGridPosition();
        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);
    }


    private void DistanceFromStartingPoint()
    {
        int newDistance = PathFinding.Instance.CalculateDistance(thisTurnStartingGridPosition, thisTurnEndridPosition);

        int delta = newDistance - distance;
        if (Mathf.Abs(delta) < 10) return;
        if (delta != 0)
        {
            unit.RegenCoverOnMove(delta);
        }

        distance = newDistance;
    }

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

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

        const int COST_PER_TILE = 10;
        int moveBudgetCost = maxMoveDistance * COST_PER_TILE;

        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);
            }
        }

        var links = PathFinding.Instance.GetPathfindingLinks();
        if (links != null && links.Count > 0)
        {
            foreach (var link in links)
            {
                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);
                        }
                    }
                }
            }

            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);
                        }
                    }
                }
            }
        }
    }

    foreach (var test in candidates)
    {
        if (!LevelGrid.Instance.IsValidGridPosition(test)) continue;
        if (test == unitPos) continue;
        if (LevelGrid.Instance.HasAnyUnitOnGridPosition(test)) continue;
        if (!PathFinding.Instance.IsWalkableGridPosition(test)) continue;

        int lowerBound = PathFinding.Instance.CalculateDistance(unitPos, test);
        if (lowerBound > moveBudgetCost) continue;

        if (!TryGetPathCostCached(unitPos, test, out int pathCost)) continue;
        if (pathCost > moveBudgetCost) continue;
```

```
            valid.Add(test);
        }

        return valid;
    }

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

    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;
    }

    private Dictionary<PathQuery, PathCacheEntry> _pathCache = new Dictionary<PathQuery, PathCacheEntry>(256);
    private int _cacheFrame = -1;

    private bool TryGetPathCostCached(GridPosition start, GridPosition end, out int cost)
    {
        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;
        }

        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,
        };
    }

    /// Serveri toiminnot

}
```

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

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

public class OverwatchAction : BaseAction
{
    private enum State
    {
        BeginOverwatchIntent,
        OverwatchIntentReady,
    }

    private State state;
    public Vector3 TargetWorld { get; private set; }

    private float stateTimer;
    private WeaponDefinition weapon;

    [SyncVar] public bool Overwatch = false;

    protected override void Awake()
    {
        base.Awake();
        var shootAction = GetComponent<ShootAction>();
        if (shootAction != null)
        {
            weapon = shootAction.GetWeapon();
        }
    }

    private void Update()
    {
        if (!isActive)
        {
            return;
        }
        stateTimer -= Time.deltaTime;
        switch (state)
        {
            case State.BeginOverwatchIntent:
                if (RotateTowards(TargetWorld, 750))
                {
                    stateTimer = 0;
                }
                break;
            case State.OverwatchIntentReady:
                break;
        }
```

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

    private void NextState()
    {
        switch (state)
        {
            case State.BeginOverwatchIntent:
                state = State.OverwatchIntentReady;
                stateTimer = 0.5f;
                break;

            case State.OverwatchIntentReady:
                ActionComplete();

                Vector3 dir = TargetWorld - unit.transform.position;
                dir.y = 0f;
                Vector3 facingWorld = (dir.sqrMagnitude > 1e-4f) ? dir.normalized : unit.transform.forward;

                var settings = GetOverwatchSettings();

                unit.GetComponent<UnitVision>().ShowUnitOverWatchVision(facingWorld, settings.coneAngleDeg);

                if (NetworkSync.IsOffline || NetworkServer.active)
                {
                    Overwatch = true;
                    PushOverwatchFacingServerLocal(facingWorld, settings);
                }
                else if (NetworkClient.active && NetworkSyncAgent.Local != null)
                {
                    var ni = unit.GetComponent<NetworkIdentity>();
                    if (ni != null)
                    {
                        NetworkSyncAgent.Local.CmdSetOverwatch(ni.netId, true, facingWorld.x, facingWorld.z);
                    }
                }
                break;
        }
    }

    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        TargetWorld = LevelGrid.Instance.GetWorldPosition(gridPosition);
        state = State.BeginOverwatchIntent;
        stateTimer = 0.7f;
        ActionStart(onActionComplete);
    }

    public override string GetActionName()
```

```
    {
        return "Overwatch";
    }

    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 0;
    }

    public bool IsOverwatch()
    {
        return Overwatch;
    }

    public void CancelOverwatchIntent()
    {
        GridSystemVisual.Instance.RemovePersistentOverwatch(unit);

        if (NetworkSync.IsOffline || NetworkServer.active)
        {
            Overwatch = false;
        }
        else if (NetworkClient.active && NetworkSyncAgent.Local != null)
        {
            var ni = unit.GetComponent<NetworkIdentity>();
            if (ni != null)
            {
                Vector3 dir = TargetWorld - unit.transform.position; dir.y = 0f;
                Vector3 facingWorld = (dir.sqrMagnitude > 1e-4f) ? dir.normalized : unit.transform.forward;
                NetworkSyncAgent.Local.CmdSetOverwatch(ni.netId, false, facingWorld.x, facingWorld.z);
            }
        }
    }
```

```
public void FacingDir()
{
    Vector3 dir = TargetWorld - unit.transform.position;
    dir.y = 0f;
    Vector3 facingWorld = (dir.sqrMagnitude > 0.0001f) ? dir.normalized : unit.transform.forward;

    var settings = GetOverwatchSettings();
    unit.GetComponent<UnitVision>()?.ShowUnitOverWatchVision(facingWorld, settings.coneAngleDeg);
}

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

// Tämä metodi tallentaa arvot payloadiin, josta saadaan jatkossa facingworld, overwach kulma säde ja etäisyys.
private void PushOverwatchFacingServerLocal(Vector3 facingWorld, OverwatchShootingSettings settings)
{
    if (NetworkServer.active || NetworkSync.IsOffline)
    {
        if (TryGetComponent<UnitStatusController>(out var status))
        {
            status.AddOrUpdate(UnitStatusType.Overwatch, new OverwatchPayload
            {
                facingWorld = OverwatchHelpers.NormalizeFacing(facingWorld), // ← Normalisoi tallennettaessa
                coneAngleDeg = settings.coneAngleDeg,
                rangeTiles = settings.rangeTiles
            });
        }
    }
}

[Server]
public void ServerApplyOverwatch(bool enabled, Vector2 facingXZ)
{
    var dir = new Vector3(facingXZ.x, 0f, facingXZ.y);
    dir = OverwatchHelpers.NormalizeFacing(dir); // ← Normalisoi heti

    bool sameState = Overwatch == enabled;
    bool sameDir = false;

    if (TryGetComponent<UnitStatusController>(out var s) &&
        s.TryGet<OverwatchPayload>(UnitStatusType.Overwatch, out var oldP))
    {
        sameDir = Vector3.Dot(oldP.facingWorld, dir) > 0.999f;
    }
```

```
        if (sameState && sameDir) return;

        Overwatch = enabled;

        if (enabled)
        {
            var settings = GetOverwatchSettings();
            s.AddOrUpdate(UnitStatusType.Overwatch, new OverwatchPayload
            {
                facingWorld = dir,
                coneAngleDeg = settings.coneAngleDeg,
                rangeTiles = settings.rangeTiles
            });
        }
        else
        {
            s.Remove(UnitStatusType.Overwatch);
        }
    }

    public OverwatchShootingSettings GetOverwatchSettings()
    {
        return weapon != null ? weapon.overwatch : OverwatchShootingSettings.Default;
    }
}
```

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

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

public class ShootAction : BaseAction
{
    public static event EventHandler<OnShootEventArgs> OnAnyShoot;
    public event EventHandler<OnShootEventArgs> OnShoot;

    [SyncVar] private bool isOverwatchShot;
    private Vector3 lastOverwatchFacing;

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

    private enum State
    {
        Aiming,
        Shooting,
        Cooloff
    }

    [SerializeField] private LayerMask obstaclesLayerMask;
    private State state;
    [SerializeField] private WeaponDefinition weapon;

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

    private int currentBurstCount;
    private int maxBurstCount;

    private float CurrentTurnSpeed        => isOverwatchShot ? weapon.overwatch.turnSpeed        : weapon.normalShooting.turnSpeed;
    private float CurrentMinAimTime       => isOverwatchShot ? weapon.overwatch.minAimTime       : weapon.normalShooting.minAimTime;
    private float CurrentAimingStateTime  => isOverwatchShot ? weapon.overwatch.aimingStateTime  : weapon.normalShooting.aimingStateTime;
    private float CurrentCooloffStateTime => isOverwatchShot ? weapon.overwatch.cooloffStateTime : weapon.normalShooting.cooloffStateTime;

    public WeaponDefinition GetWeapon() => weapon;

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

        stateTimer -= Time.deltaTime;
```

```
        switch (state)
        {
            case State.Aiming:
                if (targetUnit != null)
                {
                    if (isOverwatchShot && !IsTargetStillValid())
                    {
                        CancelShot();
                        return;
                    }

                    if (RotateTowards(targetUnit.GetWorldPosition(), CurrentTurnSpeed))
                    {
                        stateTimer = Mathf.Min(stateTimer, CurrentMinAimTime);
                    }

                    if (isOverwatchShot)
                    {
                        UpdateOverwatchConeIfRotated();
                    }
                }
                break;
            case State.Shooting:
                if (canShootBullet)
                {

                    if (targetUnit == null)
                    {
                        state = State.Cooloff;
                        stateTimer = 0.1f;
                        canShootBullet = false;
                        return;
                    }

                    Shoot();
                    canShootBullet = false;
                }
                break;
            case State.Cooloff:
                break;
        }

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

    private void UpdateOverwatchConeIfRotated()
    {
        Vector3 currentFacing = unit.transform.forward;
        currentFacing.y = 0f;
```

```
        if (currentFacing.sqrMagnitude < 1e-6f) return;
        currentFacing.Normalize();

        if (lastOverwatchFacing == Vector3.zero)
        {
            lastOverwatchFacing = currentFacing;
            return;
        }

        float dot = Vector3.Dot(lastOverwatchFacing, currentFacing);

        if (dot < 0.999f)
        {
            lastOverwatchFacing = currentFacing;
            OverwatchVisionUpdater.UpdateVision(unit, currentFacing, weapon.overwatch.coneAngleDeg);
        }
    }

    // Estää ampumasta läpi seinien jos kohde liikkuu samaan aikaan kun tätä yritetään ampua.
    private bool IsTargetStillValid()
    {
        if (targetUnit == null || targetUnit.IsDead() || targetUnit.IsDying())
            return false;

        if (!unit.TryGetComponent<UnitVision>(out var vision) || !vision.IsInitialized)
            return false;

        var targetPos = targetUnit.GetGridPosition();
        var personalVision = vision.GetUnitVisionGrids();
        if (personalVision == null || !personalVision.Contains(targetPos))
            return false;

        if (unit.TryGetComponent<UnitStatusController>(out var status) &&
            status.TryGet<OverwatchPayload>(UnitStatusType.Overwatch, out var payload))
        {
            Vector3 facingWorld = payload.facingWorld;
            float coneDeg = payload.coneAngleDeg;

            if (!vision.IsTileInCone(facingWorld, coneDeg, targetPos))
                return false;
        }

        return true;
    }

    private void CancelShot()
    {
        targetUnit = null;
        state = State.Cooloff;
        stateTimer = 0.1f;
        canShootBullet = false;
        ActionComplete();
```

```
    }

    private void NextState()
    {
        switch (state)
        {
            case State.Aiming:
                if (targetUnit == null)
                {
                    ActionComplete();
                    return;
                }

                if (isOverwatchShot && !IsTargetStillValid())
                {
                    CancelShot();
                    return;
                }

                state = State.Shooting;
                currentBurstCount = 0;
                maxBurstCount = weapon.GetRandomBurstCount();

                var unitAnimator = GetComponent<UnitAnimator>();
                if (unitAnimator != null)
                {
                    unitAnimator.NotifyBurstStart(maxBurstCount);
                }

                float shootingStateTime = 0.1f;
                stateTimer = shootingStateTime;
                break;
            case State.Shooting:
                currentBurstCount++;

                if (targetUnit == null)
                {
                    state = State.Cooloff;
                    stateTimer = 0.1f;
                    return;
                }

                if (currentBurstCount < maxBurstCount)
                {
                    canShootBullet = true;
                    stateTimer = weapon.burstShotDelay;
                }
                else
                {
                    state = State.Cooloff;
                    stateTimer = CurrentCooloffStateTime;
                }
```

```
                break;
            case State.Cooloff:
                ActionComplete();
                break;
        }
    }

    public void MarkAsOverwatchShot(bool overwatchShot)
    {

        isOverwatchShot = overwatchShot;
        if (isOverwatchShot)
        {
            lastOverwatchFacing = Vector3.zero;
        }
    }

    private void Shoot()
    {
        Debug.Log("IsOverwatchShot: " + isOverwatchShot);
        if (targetUnit == null)
        {
            return;
        }

        var result = ShootingResolver.Resolve(unit, targetUnit, weapon, isOverwatchShot);

      //  var result = ShootingResolver.Resolve(unit, targetUnit, weapon);

        OnAnyShoot?.Invoke(this, new OnShootEventArgs
        {
            targetUnit = targetUnit,
            shootingUnit = unit,
            shotTier = result.tier
        });

        OnShoot?.Invoke(this, new OnShootEventArgs
        {
            targetUnit = targetUnit,
            shootingUnit = unit,
            shotTier = result.tier
        });

        switch (result.tier)
        {
            case ShotTier.CritMiss:
                return;

            case ShotTier.Close:
                ApplyHit(result.damage, result.bypassCover, result.coverOnly, targetUnit);
                return;
```

```
            case ShotTier.Graze:
                if (GetCoverType(targetUnit) == CoverService.CoverType.None)
                {
                    MakeDamage(result.damage + weapon.NoCoverDamageBonus, targetUnit);
                    return;
                }

                ApplyHit(result.damage, result.bypassCover, result.coverOnly, targetUnit);
                return;

            case ShotTier.Hit:
                if (GetCoverType(targetUnit) == CoverService.CoverType.None)
                {
                    MakeDamage(result.damage + weapon.NoCoverDamageBonus, targetUnit);
                    return;
                }

                ApplyHit(result.damage, result.bypassCover, result.coverOnly, targetUnit);
                return;

            case ShotTier.Crit:
                if (GetCoverType(targetUnit) == CoverService.CoverType.None)
                {
                    MakeDamage(result.damage + weapon.NoCoverDamageBonus, targetUnit);
                    return;
                }
                ApplyHit(result.damage, result.bypassCover, result.coverOnly, targetUnit);
                return;
        }
    }

    public void ResetOverwatchShotState()
    {
        isOverwatchShot = false;
    }

    public override int GetActionPointsCost()
    {
        return 1;
    }

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

    public List<GridPosition> GetValidActionGridPositionList(GridPosition unitGridPosition)
    {
        var res = new List<GridPosition>();
        int r = weapon.maxShootRange;

        var cfg = LoSConfig.Instance;
```

```
        foreach (var enemy in EnumerateEnemyCandidatesInRange(unitGridPosition, r))
        {
            var gp = enemy.GetGridPosition();
            if (!RaycastVisibility.HasLineOfSightRaycastHeightAware(
                unitGridPosition, gp,
                cfg.losBlockersMask, cfg.eyeHeight, cfg.samplesPerCell, cfg.insetWU))
                continue;

            res.Add(gp);
        }
        return res;
    }

    private IEnumerable<Unit> EnumerateEnemyCandidatesInRange(GridPosition origin, int range)
    {
        bool shooterIsEnemy = unit.IsEnemy();
        foreach (var u in UnitManager.Instance.GetAllUnitList())
        {
            if (u == null) continue;
            if (u.IsEnemy() == shooterIsEnemy) continue;
            var gp = u.GetGridPosition();
            if (gp.floor != origin.floor) continue;
            int cost = SircleCalculator.Sircle(gp.x - origin.x, gp.z - origin.z);
            if (cost > 10 * range) continue;
            yield return u;
        }
    }

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

        targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPosition);

        state = State.Aiming;
        stateTimer = CurrentAimingStateTime;

        canShootBullet = true;

        ActionStart(onActionComplete);
    }

    public Unit GetTargetUnit()
    {
        return targetUnit;
    }

    public int GetMaxShootDistance()
    {
        return weapon.maxShootRange;
    }

    public override List<GridPosition> GetValidGridPositionList()
```

```
    {
        GridPosition unitGridPosition = unit.GetGridPosition();
        return GetValidActionGridPositionList(unitGridPosition);
    }

    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),
        };
    }

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

Assets/scripts/Units/UnitActions/UnitActionSystem.cs

```
using System;
using UnityEngine;
using UnityEngine.EventSystems;
using static GridSystemVisual;

/// <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()
    {
        Unit.OnAnyUnitDead += Unit_OnAnyUnitDead;
    }

    private void OnDestroy()
    {
        Unit.OnAnyUnitDead -= Unit_OnAnyUnitDead;
```

```
    }

    private void Unit_OnAnyUnitDying(object sender, EventArgs e)
    {
        if (isBusy) ClearBusy();
    }

    private void Unit_OnAnyUnitDead(object sender, EventArgs e)
    {

        var dead = sender as Unit;

        // Vapauta UI AINA
        if (isBusy) ClearBusy();

        // Jos kuollut oli valittuna, nollaa valinnat
        if (dead == selectedUnit)
        {
            ResetSelectedUnit();
            ResetSelectedAction();
        }

        CameraThaw.Thaw("Unit_OnAnyUnitDead - selected unit died");
    }

    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()
    {
        if (selectedUnit == null || selectedAction == null) return;

        if (InputManager.Instance.IsMouseButtonDownThisFrame() && selectedAction is GranadeAction)
        {
            if (selectedUnit.GetGrenadePCS() <= 0) return;
        }
```

```
        GridPosition targetGridPosition;

        if (InputManager.Instance.IsMouseButtonDownThisFrame() && selectedAction is ShootAction)
        {
            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 (unit.IsEnemy())
                    {
                        targetGridPosition = unit.GetGridPosition();
                        TryExecuteSelectedAction(targetGridPosition);
                    }
                }
            }
        }
        else if (InputManager.Instance.IsMouseButtonDownThisFrame())
        {
            Vector3 world = MouseWorld.GetPositionOnlyHitVisible();
            targetGridPosition = LevelGrid.Instance.GetGridPosition(world);
            TryExecuteSelectedAction(targetGridPosition);
        }
    }

    private void TryExecuteSelectedAction(GridPosition gp)
    {

        int steps = selectedUnit.GetMaxMoveDistance();
        int moveBudgetCost = PathFinding.CostFromSteps(steps);
        int estCost = PathFinding.Instance.CalculateDistance(selectedUnit.GetGridPosition(), gp);
        if (estCost > moveBudgetCost * 10) return;

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

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

    /// <summary>
    //      Prevents the player from performing multiple actions at the same time
    /// </summary>
    public 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())
        {
            if(selectedAction is ShootAction)
            {
                HandleSelectedAction();
            }
            return;
        }
        selectedUnit = unit;
        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;
    }

    public void ResetSelectedAction()
    {
        selectedAction = null;
    }

    public void ResetSelectedUnit()
    {
        selectedUnit = null;
    }

    // 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;
using System.Collections;

[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;

    [Header("Visual Effects")]
    [SerializeField] private GameObject muzzleFlashPrefab;
    [SerializeField] private float muzzleFlashDuration = 0.1f;

    [Header("Audio")]
    [SerializeField] private AudioSource weaponAudioSource;
    [SerializeField] private AudioSource tailAudioSource; // ← LISÄÄ TÄMÄ
    [SerializeField] private AudioClip[] rifleShootVariations;
    [SerializeField] private AudioClip rifleShootTail; // ← Muuta nimi selkeämmäksi

    [Header("Audio Settings")]
    [SerializeField] private float pitchVariation = 0.1f;
    [SerializeField] private float volumeVariation = 0.15f;
    [SerializeField] private float baseVolume = 1f;
    [SerializeField] private float maxHearingDistance = 50f;
    [SerializeField] private AnimationCurve volumeRolloff = AnimationCurve.Linear(0, 1, 1, 0);

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

    private MoveAction _move;
    private ShootAction _shoot;
    private GranadeAction _grenade;
    private MeleeAction _melee;

    private bool useNetwork;
```

```
    private HealthSystem hs;

    private int currentShotInBurst = 0;
    private int totalShotsInBurst = 0;

    private void Awake()
    {
        if (!animator) animator = GetComponent<Animator>();
        if (!netAnim) netAnim = GetComponent<NetworkAnimator>();

        useNetwork = NetMode.IsOnline
             && netAnim != null
             && (isServer || isOwned);

        TryGetComponent(out _move);
        TryGetComponent(out _shoot);
        TryGetComponent(out _grenade);
        TryGetComponent(out _melee);
        TryGetComponent(out hs);

        SetupAudioSource();
    }

    private void SetupAudioSource()
    {
        if (weaponAudioSource == null) return;

        weaponAudioSource.spatialBlend = 1f;
        weaponAudioSource.rolloffMode = AudioRolloffMode.Custom;
        weaponAudioSource.maxDistance = maxHearingDistance;
        weaponAudioSource.minDistance = 1f;
        weaponAudioSource.dopplerLevel = 0f;
        weaponAudioSource.spread = 0f;
        weaponAudioSource.SetCustomCurve(AudioSourceCurveType.CustomRolloff, volumeRolloff);

        // Aseta tail AudioSource samoilla asetuksilla
        if (tailAudioSource != null)
        {
            tailAudioSource.spatialBlend = 1f;
            tailAudioSource.rolloffMode = AudioRolloffMode.Custom;
            tailAudioSource.maxDistance = maxHearingDistance;
            tailAudioSource.minDistance = 1f;
            tailAudioSource.dopplerLevel = 0f;
            tailAudioSource.spread = 0f;
            tailAudioSource.SetCustomCurve(AudioSourceCurveType.CustomRolloff, volumeRolloff);
        }
    }

    private void OnEnable()
    {
        if (_move)
```

```
        {
            _move.OnStartMoving -= MoveAction_OnStartMoving;
            _move.OnStopMoving -= MoveAction_OnStopMoving;
            _move.OnStartMoving += MoveAction_OnStartMoving;
            _move.OnStopMoving += MoveAction_OnStopMoving;
        }

        if (_shoot)
        {
            _shoot.OnShoot -= ShootAction_OnShoot;
            _shoot.OnShoot += ShootAction_OnShoot;
        }

        if (_grenade)
        {
            _grenade.ThrowGranade -= GrenadeAction_ThrowGranade;
            _grenade.ThrowReady -= GrenadeAction_ThrowReady;
            _grenade.ThrowGranade += GrenadeAction_ThrowGranade;
            _grenade.ThrowReady += GrenadeAction_ThrowReady;
        }

        if (_melee)
        {
            _melee.OnMeleeActionStarted -= MeleeAction_OnMeleeActionStarted;
            _melee.OnMeleeActionCompleted -= MeleeAction_OnMeleeActionCompleted;
            _melee.OnMeleeActionStarted += MeleeAction_OnMeleeActionStarted;
            _melee.OnMeleeActionCompleted += MeleeAction_OnMeleeActionCompleted;
        }

        if (hs != null)
            hs.OnDying += OnDying_StopSending;
    }

    private void OnDisable()
    {
        if (_move)
        {
            _move.OnStartMoving -= MoveAction_OnStartMoving;
            _move.OnStopMoving  -= MoveAction_OnStopMoving;
        }
        if (_shoot)
        {
            _shoot.OnShoot -= ShootAction_OnShoot;
        }
        if (_grenade)
        {
            _grenade.ThrowGranade -= GrenadeAction_ThrowGranade;
            _grenade.ThrowReady   -= GrenadeAction_ThrowReady;
        }
        if (_melee)
        {
            _melee.OnMeleeActionStarted -= MeleeAction_OnMeleeActionStarted;
```

```
            _melee.OnMeleeActionCompleted -= MeleeAction_OnMeleeActionCompleted;
        }

        if (hs != null)
            hs.OnDying -= OnDying_StopSending;
    }

    private void Start()
    {
        EquipRifle();
    }

    public void SetTrigger(string name)
    {
        if (useNetwork) netAnim.SetTrigger(name);
        else animator.SetTrigger(name);
    }

    private void OnDying_StopSending(object s, EventArgs e)
    {
        pendingGrenadeAction = null;
        useNetwork = false;
    }

    private void MoveAction_OnStartMoving(object sender, EventArgs e)
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        animator.SetBool("IsRunning", true);
    }
    private void MoveAction_OnStopMoving(object sender, EventArgs e)
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        animator.SetBool("IsRunning", false);
    }

    public Transform ShootPoint => shootPointTransform;
    public GameObject BulletPrefab => bulletProjectilePrefab;

    public void NotifyBurstStart(int burstSize)
    {
        currentShotInBurst = 0;
        totalShotsInBurst = burstSize;
    }

    public void PlayRifleShootEffects()
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        if (weaponAudioSource == null) return;

        currentShotInBurst++;

        // MUZZLE FLASH
```

```
        if (muzzleFlashPrefab != null && shootPointTransform != null)
        {
            GameObject flash = Instantiate(muzzleFlashPrefab, shootPointTransform.position, shootPointTransform.rotation);
            Destroy(flash, muzzleFlashDuration);
        }

        // LAUKAISUÄÄNI
        if (rifleShootVariations != null && rifleShootVariations.Length > 0)
        {
            AudioClip shotClip = rifleShootVariations[UnityEngine.Random.Range(0, rifleShootVariations.Length)];

            float pitch = 1f + UnityEngine.Random.Range(-pitchVariation, pitchVariation);
            float volume = baseVolume + UnityEngine.Random.Range(-volumeVariation, volumeVariation);

            weaponAudioSource.pitch = pitch;
            weaponAudioSource.PlayOneShot(shotClip, volume);
            weaponAudioSource.pitch = 1f;
        }

        // TAIL-ÄÄNI (erillisellä AudioSourcella, soitetaan samanaikaisesti tai pienellä viiveellä)
        if (tailAudioSource != null && rifleShootTail != null)
        {
            float tailPitch = 1f + UnityEngine.Random.Range(-pitchVariation * 0.3f, pitchVariation * 0.3f);
            float tailVolume = baseVolume * 0.7f + UnityEngine.Random.Range(-volumeVariation * 0.5f, volumeVariation * 0.5f);

            tailAudioSource.pitch = tailPitch;
            tailAudioSource.PlayOneShot(rifleShootTail, tailVolume);
            tailAudioSource.pitch = 1f;
        }
    }

    private void ShootAction_OnShoot(object sender, ShootAction.OnShootEventArgs e)
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;

        if (e.targetUnit == null)
        {
            return;
        }

        SetTrigger("Shoot");
        PlayRifleShootEffects();

        Vector3 target = BulletTargetCalculator.CalculateBulletTarget(
            e.targetUnit,
            e.shotTier,
            e.shootingUnit
        );

        // Määritä pitikö osua Unittiin
        bool shouldHitUnit = e.shotTier != ShotTier.CritMiss && e.shotTier != ShotTier.Close;
```

```
        if (NetMode.IsOnline)
        {
            NetworkSync.SpawnBullet(bulletProjectilePrefab, shootPointTransform.position, target, shouldHitUnit, this.GetActorId());
        }
        else
        {
            OfflineGameSimulator.SpawnBullet(bulletProjectilePrefab, shootPointTransform.position, target, shouldHitUnit);
        }
    }


    private void MeleeAction_OnMeleeActionStarted(object sender, EventArgs e)
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        EquipMelee();
        SetTrigger("Melee");
    }

    private void MeleeAction_OnMeleeActionCompleted(object sender, EventArgs e)
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        EquipRifle();
    }

    private void GranadeActionStart()
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        weaponVis.OwnerRequestSet(rifleRight: false, rifleLeft: true, meleeLeft: false, grenade: false);
    }

    private Vector3 pendingGrenadeTarget;
    private GranadeAction pendingGrenadeAction;

    private void GrenadeAction_ThrowGranade(object sender, EventArgs e)
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        pendingGrenadeAction = (GranadeAction)sender;
        pendingGrenadeTarget = pendingGrenadeAction.TargetWorld;

        GranadeActionStart();
        SetTrigger("ThrowGrenade");

    }

    public void AE_PickGrenadeStand()
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        EguipGranade();
    }

    public Transform ThrowPoint => rightHandTransform;
    public GameObject GrenadePrefab => granadeProjectilePrefab;
```

```
    public void AE_ThrowGrenadeStandRelease()
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        if (pendingGrenadeAction == null) return;

        if (NetworkClient.active || NetworkServer.active)
        {
            var ni = GetComponentInParent<NetworkIdentity>();
            if (!(isClient && ni && ni.isOwned)) return;
        }

        Vector3 origin = rightHandTransform.position;

        float farWU = pendingGrenadeAction.GetMaxThrowRangeWU();

        if (NetMode.IsOnline)

            NetworkSync.SpawnGrenade(granadeProjectilePrefab, origin, pendingGrenadeTarget, farWU, this.GetActorId());

        else
        {
            OfflineGameSimulator.SpawnGrenade(granadeProjectilePrefab, origin, pendingGrenadeTarget, farWU);
        }

        pendingGrenadeAction?.OnGrenadeBehaviourComplete();
        pendingGrenadeAction = null;
    }

    public void AE_OnGrenadeThrowStandFinished()
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        EquipRifle();
    }

    private void GrenadeAction_ThrowReady(object sender, EventArgs e)
    {
       weaponVis.OwnerRequestSet(rifleRight: false, rifleLeft: true, meleeLeft: false, grenade: false);
    }

    private void EquipRifle()
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        weaponVis.OwnerRequestSet(rifleRight: true, rifleLeft: false, meleeLeft: false, grenade: false);
    }
    private void EquipMelee()
    {
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        weaponVis.OwnerRequestSet(rifleRight: true, rifleLeft: false, meleeLeft: true, grenade: false);
    }
    private void EguipGranade()
    {
```

```
        if (hs && (hs.IsDying() || hs.IsDead())) return;
        weaponVis.OwnerRequestSet(rifleRight: false, rifleLeft: true, meleeLeft: false, grenade: true);
    }

    public Transform GetrightHandTransform()
    {
        return rightHandTransform;
    }
}
```

Assets/scripts/Units/UnitAnimatorEventRelay.cs

```
using UnityEngine;

public class AnimationEventRelay : MonoBehaviour
{
    [SerializeField] private UnitAnimator unitAnimator;

    void Awake()
    {
        if (!unitAnimator) unitAnimator = GetComponentInParent<UnitAnimator>();
    }

    public void AE_ThrowGrenadeStandRelease()
    {
        unitAnimator?.AE_ThrowGrenadeStandRelease();
    }

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

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

Assets/scripts/Units/UnitArchetypes/UnitArchetype.cs

```
using UnityEngine;

[CreateAssetMenu(menuName = "RogueShooter/UnitArchetype")]
public class UnitArchetype : ScriptableObject
{

    [Header("BASE SKILLS")]
    [Space(10)]
    [Header("Covering Skill")]
    public int personalCoverMax = 200;
    public int coverRegenOnMove = 1;
    public int coverRegenPerUnusedAP = 25;

    [Tooltip("Kuinka paljon matala suoja vähentää vihollisen todennäköisyyttä osua. Huom! Suhteellinen vähennys. Esim 2% vähentää 100% taidon 98%, ja 50% taidon 49%")]
    public int LowCoverEnemyHitPenalty = 12;      // -12% vihollisen osumatodennäköisyyteen
    [Tooltip("Kuinka paljon korkea suoja vähentää vihollisen todennäköisyyttä osua. Huom! Suhteellinen vähennys. Esim 2% vähentää 100% taidon 98%, ja 50% taidon 49%")]
    public int highCoverEnemyHitPenalty = 25;      // -25% vihollinen osumatodennäköisyyteen

    [Space(10)]
    [Header("Moving Skill")]
    public int moveRange = 4;

     [Tooltip("Kuinka paljon liike vähentää vihollisen mahdollisuutta osua. Huom! Suhteellinen vähennys. Esim 2% vähentää 100% taidon 98%, ja 50% taidon 49%")]
    public int moveEnemyHitPenalty = 5; // Kuinka paljon vähentää osumatodennäköisyyttä kun on liikkeessä.

    [Space(10)]
    [Header("Shooting Skill")]
    public int basicShootinSkill = 69;
    public int shootingSkillLevel = 0; // 0..10
    public int accuracyBonusPerSkillLevel = 3; // +3% tarkkuutta / taso


    [Space(10)]
    [Header("Grenade Skill")]
    public int grenadeCapacity = 2;
    public int throwingRange = 7;

    [Space(10)]
    [Header("Vision Skill")]
    public int visionRange = 20;
    public bool useHeightAware = true;


    [Header("Progression (optional)")]
    public AnimationCurve coverMaxByLevel = AnimationCurve.Linear(1, 200, 10, 300);
    public AnimationCurve regenByLevel = AnimationCurve.Linear(1, 20, 10, 35);

}
```

Assets/scripts/Units/UnitArchetypes/UnitSkills.cs

```
[System.Serializable]
public class UnitSkills
{
    public int personalCoverMax = 200;
    public int CoverRegenOnMove = 20;
    public float coverMitigationMultiplier = 1f; // varalle jatkoon (+% vaimennukseen)
}
```

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> allUnitsList;
    private List<Unit> friendlyUnitList;
    private List<Unit> enemyUnitList;

    private readonly HashSet<Unit> unitSet = new();

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

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

    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)
    {
```

```
        allUnitsList.RemoveAll(u => u == null );
        friendlyUnitList.RemoveAll(u => u == null);
        enemyUnitList.RemoveAll(u => u == null);

        // 1) Estä duplikaatit
        Unit unit = sender as Unit;
        if (!unitSet.Add(unit)) return;
        if (!allUnitsList.Contains(unit)) allUnitsList.Add(unit);

        if (GameModeManager.SelectedMode == GameMode.SinglePlayer || GameModeManager.SelectedMode == GameMode.CoOp)
        {

            if (unit.IsEnemy())
            {
                if (!enemyUnitList.Contains(unit)) enemyUnitList.Add(unit);
                unit.Team = Team.Enemy;
            }
            else
            {
                if (!friendlyUnitList.Contains(unit)) friendlyUnitList.Add(unit);
                unit.Team = Team.Player;
            }

        }
        if (GameModeManager.SelectedMode == GameMode.Versus)
        {
            if(NetworkSync.IsOwnerHost(unit.OwnerId))
            {
                friendlyUnitList.Add(unit);
                unit.Team = Team.Player;
            } else
            {
                enemyUnitList.Add(unit);
                unit.Team = Team.Enemy;
            }
        }
    }

    private void Unit_OnAnyUnitDead(object sender, EventArgs e)
    {

        Unit unit = sender as Unit;
        unitSet.Remove(unit);

        // Poista kaikki esiintymät JA siivoa nullit samalla
        allUnitsList.RemoveAll(u => u == null || u == unit);
        friendlyUnitList.RemoveAll(u => u == null || u == unit);
        enemyUnitList.RemoveAll(u => u == null || u == unit);

    }

    public void ClearNullUnits()
```

```
    {

    }


    // Yksinkertainen "puhdas" read-API
    public IReadOnlyList<Unit> GetEnemyUnitList()
    {
        enemyUnitList.RemoveAll(u => u == null);
        return enemyUnitList;
    }

    public List<Unit> GetAllUnitList()
    {
        return allUnitsList;
    }

    public List<Unit> GetFriendlyUnitList()
    {
        return friendlyUnitList;
    }

    public void ClearAllUnitLists()
    {
        allUnitsList.Clear();
        friendlyUnitList.Clear();
        enemyUnitList.Clear();
    }
}
```

Assets/scripts/Units/UnitPathFinding/PathFinding.cs

```
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 = 14;

    [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());

            }
        }
    } else
    {
        // Muuten etsi kaikki scenestä
        var allLinkMbs = FindObjectsByType<PathfindingLinkMonoBehaviour>(FindObjectsSortMode.None);
        if (allLinkMbs == null)
        {
            Debug.LogWarning("[PF] No PathfindingLinkMonoBehaviour found in scene");
            return;
        }

        foreach (var linkMb in allLinkMbs)
        {
```

```
                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);
    }

    public 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);
    */
    public void SetIsWalkableGridPosition(GridPosition gridPosition, bool isWalkable)
    {
        if (LevelGrid.Instance == null) { Debug.LogWarning("[PF] SetIsWalkable before LevelGrid"); return; }
        if (!LevelGrid.Instance.IsValidGridPosition(gridPosition))
        {
            Debug.LogWarning($"[PF] Ignore SetIsWalkable for OUT-OF-BOUNDS {gridPosition}");
            return;
        }

        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

```
public class PathfindingLink
{
    public GridPosition gridPositionA;
    public GridPosition gridPositionB;
}
```

Assets/scripts/Units/UnitPathFinding/PathfindingLinkMonoBehaviour.cs

```
using UnityEngine;

// Linkit asetetaan tyhjään linkkejä sisältävään game objektiin joka annetaan PathFindig.cs
// Pathfinding etsii yhteydet Editorissa ennakkoon annetusta linkki conteinerista.
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()
    {
        var aW = transform.TransformPoint(linkPositionA);
        var bW = transform.TransformPoint(linkPositionB);
        return new PathfindingLink
        {
            gridPositionA = LevelGrid.Instance.GetGridPosition(aW),
            gridPositionB = LevelGrid.Instance.GetGridPosition(bW),
        };
    }
}
```

Assets/scripts/Units/UnitPathFinding/PathNode.cs

```
[System.Flags]
public enum EdgeMask { None = 0, N = 1, E = 2, S = 4, W = 8 }

[System.Flags]
public enum CoverMask { 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

    private CoverMask highCover;      // täyskorkea suoja suunnittain
    private CoverMask lowCover;       // matala suoja suunnittain

    public void ClearWalls() => walls = EdgeMask.None;
    public void AddWall(EdgeMask dir) => walls |= dir;
    public bool HasWall(EdgeMask dir) => (walls & dir) != 0;


    public void ClearCover() { highCover = CoverMask.None; lowCover = CoverMask.None; }
    public void AddHighCover(CoverMask d) => highCover |= d;
    public void AddLowCover(CoverMask d) => lowCover |= d;
    public bool HasHighCover(CoverMask d) => (highCover & d) != 0;
    public bool HasLowCover(CoverMask d) => (lowCover & d) != 0;


    public CoverMask GetHighCoverMask() => highCover;
    public CoverMask GetLowCoverMask() => lowCover;

    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);
    }

}
*/
using Mirror;
using UnityEngine;
using System.Collections;

public class RagdollPoseBinder : NetworkBehaviour
{
    [SyncVar] public uint sourceUnitNetId;
    [SyncVar] public Vector3 lastHitPos;
    [SyncVar] public int overkill;

    [SerializeField] float bindTimeout = 0.75f;   // varalta jos hierarkia/NetID tulee myöhässä

    UnitRagdoll ragdoll;

    void Awake()
    {
        ragdoll = GetComponent<UnitRagdoll>();
    }

    public override void OnStartClient()
    {
        base.OnStartClient();
        StartCoroutine(Co_TryBindUntilFound());
    }

    IEnumerator Co_TryBindUntilFound()
    {
        float t = 0f;
        while (t < bindTimeout)
        {
            Transform rootBone = TryResolveOriginalRootBoneOnClient();
            if (rootBone != null)
            {
                // siirrä metadatat ja sido pose
                if (ragdoll != null)
                {
                    ragdoll.SetOverkill(overkill);
                    ragdoll.SetLastHitPosition(lastHitPos);
                    ragdoll.Setup(rootBone);
                }
                yield break;
            }
```

```
            t += Time.deltaTime;
            yield return null;
        }

        Debug.LogWarning("[RagdollPoseBinder] Failed to bind original root bone in time.");
    }

    Transform TryResolveOriginalRootBoneOnClient()
    {
        if (!NetworkClient.active) return null;
        if (!NetworkClient.spawned.TryGetValue(sourceUnitNetId, out var srcNi) || srcNi == null) return null;

        // Hae kaatuneen unitin spawneri, jossa viite on serialized
        var spawner = srcNi.GetComponentInChildren<UnitRagdollSpawn>(true);
        if (spawner != null && spawner.OriginalRagdollRootBone != null)   // (huom. kirjoitusasu)
            return spawner.OriginalRagdollRootBone;

        return 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 OnDisable()
    {
        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;

        if (NetMode.IsServer)
        {
            NetworkSync.SpawnRagdoll(
            ragdollPrefab.gameObject,
            transform.position,
            transform.rotation,
            id,
            lastHitPosition,
            overkill);
        } else
        {
            OfflineGameSimulator.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();
                return;
            }
            else
            {
                Debug.LogWarning("[UI] TurnSystem.Instance is null");
            }
        }

        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();

    }

    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; }
        }

        TurnSystem.Instance.NextTurn();
    }


    // ====== 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()
    {
        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;
using System;

/// <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;

     // --- UUTTA: kulmabadge
    [Header("Corner badge (optional)")]
    [SerializeField] private RectTransform cornerRoot;     // drag: CornerBadge
    [SerializeField] private TextMeshProUGUI cornerText;   // drag: CornerText

    private BaseAction baseAction;

    public void SetBaseAction(BaseAction baseAction)
    {
        this.baseAction = baseAction;
        textMeshPro.text = baseAction.GetActionName().ToUpper();

        actionButton.onClick.AddListener(() =>
        {
            UnitActionSystem.Instance.SetSelectedAction(baseAction);
        });

        RefreshCorner();
    }

    void OnEnable()
    {
        TrySub(true);
        RefreshCorner();
    }

    void OnDisable()
    {
        TrySub(false);
    }

    private void TrySub(bool on)
    {
        // turvalliset unsub/sub -kutsut
        if (UnitActionSystem.Instance != null)
```

```
        {
            if (on)
            {
                UnitActionSystem.Instance.OnSelectedUnitChanged   += OnUiRefresh;
                UnitActionSystem.Instance.OnSelectedActionChanged += OnUiRefresh;
                UnitActionSystem.Instance.OnActionStarted         += OnUiRefresh;

                // HUOM: tämä on EventHandler<bool>, EI EventHandler
                UnitActionSystem.Instance.OnBusyChanged           += OnBusyChanged;
            }
            else
            {
                UnitActionSystem.Instance.OnSelectedUnitChanged   -= OnUiRefresh;
                UnitActionSystem.Instance.OnSelectedActionChanged -= OnUiRefresh;
                UnitActionSystem.Instance.OnActionStarted         -= OnUiRefresh;
                UnitActionSystem.Instance.OnBusyChanged           -= OnBusyChanged;
            }
        }

        if (TurnSystem.Instance != null)
        {
            if (on)  TurnSystem.Instance.OnTurnChanged += OnTurnChanged;
            else     TurnSystem.Instance.OnTurnChanged -= OnTurnChanged;
        }

        if (on)
            BaseAction.OnAnyActionStarted += OnAnyActionStarted;
        else
            BaseAction.OnAnyActionStarted -= OnAnyActionStarted;
    }


    private void OnUiRefresh(object sender, EventArgs e) => RefreshCorner();
    private void OnAnyActionStarted(object sender, EventArgs e) => RefreshCorner();
    private void OnTurnChanged(object sender, EventArgs e) => RefreshCorner();

    private void OnBusyChanged(object sender, bool isBusy) => RefreshCorner();

    private void RefreshCorner()
    {
        // Näytä kulmalaskuri vain kranaatti-napissa
        bool isGrenade = baseAction is GranadeAction;
        if (!isGrenade)
        {
            if (cornerRoot) cornerRoot.gameObject.SetActive(false);
            return;
        }

        var unit = UnitActionSystem.Instance ? UnitActionSystem.Instance.GetSelectedUnit() : null;
        int pcs = unit ? unit.GetGrenadePCS() : 0;   // Unitilla on GetGrenadePCS()

        if (cornerText) cornerText.text = pcs.ToString();
```

```
        if (cornerRoot) cornerRoot.gameObject.SetActive(true);
    }

    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 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;
    }

    void OnEnable()
    {
        if (UnitActionSystem.Instance != null)
        {
            UnitActionSystem.Instance.OnSelectedUnitChanged += UnitActionSystem_OnSelectedUnitChanged;
            UpdateVisual();
        }

        TurnSystem.Instance.OnTurnChanged += turnSystem_OnTurnChanged;
    }



    void OnDisable()
    {
        if (UnitActionSystem.Instance != null)
        {
            UnitActionSystem.Instance.OnSelectedUnitChanged -= UnitActionSystem_OnSelectedUnitChanged;
            UpdateVisual();
        }

        TurnSystem.Instance.OnTurnChanged -= turnSystem_OnTurnChanged;
    }

    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;
    }

    public void ResetSelectedVisual()
    {
        if (meshRenderer) meshRenderer.enabled = false;
    }

    private void turnSystem_OnTurnChanged(object sender, EventArgs e)
    {
        ResetSelectedVisual();
    }
}
```

Assets/scripts/Units/UnitSkills/CoverSkill.cs

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

[RequireComponent(typeof(Unit))]
public class CoverSkill : NetworkBehaviour
{
    [SerializeField] private int newCoverBonusHalf = 15;
    [SerializeField] private int newCoverBonusFull = 25;

    private int personalCover;
    private int personalCoverMax;
    private int thisTurnStartingPersonalCover;

    private int currentTurnCoverBonus;

   [SyncVar] private bool hasMoved = false;

    protected Unit unit;

    private int _lastProcessedTurnId = -1;

    public event Action<int, int> OnCoverPoolChanged;

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

    private void Start()
    {
        if (unit.archetype != null)
        {
            personalCoverMax = unit.archetype.personalCoverMax;
        }

        personalCover = personalCoverMax;
        thisTurnStartingPersonalCover = personalCover;
        currentTurnCoverBonus = 0;
        hasMoved = false;

        OnCoverPoolChanged?.Invoke(personalCover, personalCoverMax);
    }

    private void OnEnable()
    {
        TurnSystem.Instance.OnTurnStarted += OnTurnStarted_HandleTurnStarted;
        TurnSystem.Instance.OnTurnEnded += OnTurnEnded_HandleTurnEnded;
    }
```

```
    private void OnDisable()
    {
        TurnSystem.Instance.OnTurnStarted -= OnTurnStarted_HandleTurnStarted;
        TurnSystem.Instance.OnTurnEnded -= OnTurnEnded_HandleTurnEnded;
    }

    public int GetPersonalCover()
    {
        return personalCover;
    }

    public void SetPersonalCover(int value)
    {
        // KRIITTINEN: Älä lähetä komentoja kuolleelle Unitille
        if (unit != null && (unit.IsDying() || unit.IsDead())) return;

        if (!NetworkServer.active && !NetworkClient.active)
        {
            ApplyCoverLocal(value);
            return;
        }

        if (NetworkServer.active)
        {
            ApplyCoverServer(value);
            return;
        }

        var ni = GetComponent<NetworkIdentity>();
        if (NetworkClient.active && NetworkSyncAgent.Local != null && ni != null && ni.netId != 0)
        {
            NetworkSyncAgent.Local.CmdSetUnitCover(ni.netId, value);
        }
    }

    public void SetCoverBonus()
    {
        // CLIENT: pyydä serveriä tekemään
        if (NetworkClient.active && !NetworkServer.active)
        {
            var ni = GetComponent<NetworkIdentity>();
            NetworkSyncAgent.Local?.CmdApplyCoverBonus(ni.netId);
            return;
        }

        // SERVER / OFFLINE: varsinainen työ
        ServerApplyCoverBonus();
    }

    //[Server]
    public void ServerApplyCoverBonus()
```

```
    {
        // Vain Server TAI Offline saa suorittaa
        if (NetworkClient.active && !NetworkServer.active) return;

        // TÄRKEÄ: Jos on jo liikuttu tällä vuorolla, resetoi bonus ENSIN
        if (hasMoved && currentTurnCoverBonus != 0)
        {
            int newCover = personalCover - currentTurnCoverBonus;
            if (newCover < thisTurnStartingPersonalCover)
                newCover = thisTurnStartingPersonalCover;

            personalCover = newCover;
            currentTurnCoverBonus = 0;
        }
        hasMoved = true;
        if (unit.IsUnderFire)
        {
            personalCover = thisTurnStartingPersonalCover;
            OnCoverPoolChanged?.Invoke(personalCover, personalCoverMax);
            NetworkSync.UpdateCoverUI(unit);
            return;
        }

        var gp = unit.GetGridPosition();
        var pf = PathFinding.Instance;
        if (pf == null) return;

        var node = pf.GetNode(gp.x, gp.z, gp.floor);
        if (node == null) return;

        var t = CoverService.GetNodeAnyCover(node);
        int bonus = t == CoverService.CoverType.High ? newCoverBonusFull :
                    t == CoverService.CoverType.Low  ? newCoverBonusHalf : 0;

        currentTurnCoverBonus = bonus;
         if (bonus > 0)
        {
            if (unit.IsUnderFire)
            {
                SetPersonalCover(thisTurnStartingPersonalCover);
            }
            else
            {
                SetPersonalCover(personalCover + bonus);
            }

        }
        else
        {
            NetworkSync.UpdateCoverUI(unit);
        }
    }
```

```
    private void ApplyCoverLocal(int value)
    {
        personalCover = Mathf.Clamp(value, 0, personalCoverMax);
        OnCoverPoolChanged?.Invoke(personalCover, personalCoverMax);
    }

    //[Server]
    private void ApplyCoverServer(int value)
    {
        // Vain Server TAI Offline saa suorittaa
        if (NetworkClient.active && !NetworkServer.active) return;

        personalCover = Mathf.Clamp(value, 0, personalCoverMax);
        OnCoverPoolChanged?.Invoke(personalCover, personalCoverMax);
        NetworkSync.UpdateCoverUI(unit);
    }

    public void RegenCoverOnMove(int distance)
    {
        // KRIITTINEN: Älä lähetä komentoja kuolleelle Unitille
        if (unit != null && (unit.IsDying() || unit.IsDead())) return;

        if (NetworkClient.active && !NetworkServer.active)
        {
            var ni = GetComponent<NetworkIdentity>();
            if (ni != null && ni.netId != 0)
                NetworkSyncAgent.Local?.CmdRegenCoverOnMove(ni.netId, distance);
            return;
        }

        ServerRegenCoverOnMove(distance);
    }

    //[Server]
    public void ServerRegenCoverOnMove(int distance)
    {
        // Vain Server TAI Offline saa suorittaa
        if (NetworkClient.active && !NetworkServer.active) return;

        int regenPerTile = unit.archetype != null ? unit.archetype.coverRegenOnMove : 5;
        int tileDelta = distance / 10;
        int coverChange = regenPerTile * tileDelta;
        int newCover = personalCover + coverChange;

        if (newCover <= thisTurnStartingPersonalCover)
        {
            newCover = thisTurnStartingPersonalCover;
        }

        SetPersonalCover(Mathf.Clamp(newCover, 0, personalCoverMax));
    }
```

```
    public void RegenCoverBy(int amount)
    {
        if (amount == 0) return;
        SetPersonalCover(personalCover + amount);
    }

    public int GetCoverRegenPerUnusedAP()
    {
        if (!unit.IsUnderFire)
        {
            return unit.archetype != null ? unit.archetype.coverRegenPerUnusedAP : 0;
        }
        return 0;
    }

    public int GetPersonalCoverMax() => personalCoverMax;

    public float GetCoverNormalized()
    {
        return (float)personalCover / personalCoverMax;
    }

    public void ApplyNetworkCover(int current, int max)
    {
        personalCoverMax = max;
        personalCover = Mathf.Clamp(current, 0, max);
        OnCoverPoolChanged?.Invoke(personalCover, personalCoverMax);
    }

    public int GetCurrentCoverBonus()
    {
        return currentTurnCoverBonus;
    }

    public void ResetCurrentCoverBonus()
    {
        // KRIITTINEN: Älä lähetä komentoja kuolleelle Unitille
        if (unit != null && (unit.IsDying() || unit.IsDead())) return;

        if (NetworkClient.active && !NetworkServer.active)
        {
            var ni = GetComponent<NetworkIdentity>();
            if (ni != null && ni.netId != 0)
                NetworkSyncAgent.Local?.CmdResetCurrentCoverBonus(ni.netId);
            return;
        }

        ServerResetCurrentCoverBonus();
    }

    //[Server]
```

```
    public void ServerResetCurrentCoverBonus()
    {
        // Vain Server TAI Offline saa suorittaa
        if (NetworkClient.active && !NetworkServer.active) return;

        if (currentTurnCoverBonus != 0)
        {
            int newCover = personalCover - currentTurnCoverBonus;
            if (newCover < thisTurnStartingPersonalCover)
                newCover = thisTurnStartingPersonalCover; // vuoron aloitus on minimi

            SetPersonalCover(newCover);
            currentTurnCoverBonus = 0; // estää tuplavähennykset
        }
    }

    public bool HasMoved()
    {
        return hasMoved;
    }

    private void OnTurnStarted_HandleTurnStarted(Team startTurnTeam, int turnId)
    {
        if (NetworkClient.active && !NetworkServer.active) return;
        // Vain oman puolen vuorolla
        if (unit.Team != startTurnTeam) return;

        // (Valinnainen) suoja duplikaateilta, jos eventti laukeaa useammin samassa vuorossa
        if (_lastProcessedTurnId == turnId) return;
        _lastProcessedTurnId = turnId;

        // Resetit vain omalle puolelle:
        thisTurnStartingPersonalCover = personalCover;
        currentTurnCoverBonus = 0;
        hasMoved = false;
    }

    private void OnTurnEnded_HandleTurnEnded(Team endTurnTeam, int turnId)
    {
        if (NetworkClient.active && !NetworkServer.active) return;
        // Vain kun toisen vuoro alkaa. Unitit eivät ole enää tulen alla.
        if (unit.Team != endTurnTeam) return;
        unit.SetUnderFire(false);
    }
}
```

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;

/// <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;

    [SerializeField] private Image personalCoverBarImage;

    /// <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()
    {

        Unit.OnAnyActionPointsChanged += Unit_OnAnyActionPointsChanged;
        healthSystem.OnDamaged += HealthSystem_OnDamaged;
        unit.OnCoverPoolChanged += Unit_OnCoverPoolChanged;

        UpdateActionPointsText();
```

```
        UpdateHealthBarUI();
        Unit_OnCoverPoolChanged(unit.GetPersonalCover(), unit.GetPersonalCoverMax());


        // 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 OnDisable()
    {
        Unit.OnAnyActionPointsChanged -= Unit_OnAnyActionPointsChanged;
        healthSystem.OnDamaged -= HealthSystem_OnDamaged;
        PlayerLocalTurnGate.LocalPlayerTurnChanged -= PlayerLocalTurnGate_LocalPlayerTurnChanged;
        unit.OnCoverPoolChanged -= Unit_OnCoverPoolChanged;
    }

    private void OnDestroy()
    {
        Unit.OnAnyActionPointsChanged -= Unit_OnAnyActionPointsChanged;
        healthSystem.OnDamaged -= HealthSystem_OnDamaged;
        PlayerLocalTurnGate.LocalPlayerTurnChanged -= PlayerLocalTurnGate_LocalPlayerTurnChanged;
        unit.OnCoverPoolChanged -= Unit_OnCoverPoolChanged;
    }

    private void UpdateActionPointsText()
    {
        actionPointsText.text = unit.GetActionPoints().ToString();
    }

    private void Unit_OnAnyActionPointsChanged(object sender, EventArgs e)
    {
        UpdateActionPointsText();
    }

    private void UpdateCoverBarUI()
    {
        personalCoverBarImage.fillAmount = unit.GetCoverNormalized();
    }
```

```
    private void Unit_OnCoverPoolChanged(int current, int max)
    {

        UpdateCoverBarUI();
    }

    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/Units/UnitStatus/StatusCoordinator.cs

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

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

    [Header("Enemy Overwatch Display")]
    [SerializeField] private float enemyOverwatchVisibilityDuration = 3f;

    private readonly Dictionary<Unit, float> _nextReactAt = new();
    private readonly Dictionary<int, HashSet<Unit>> overwatchByTeam = new();

    [SerializeField] private bool onlyOneOverwatchAttackPerMovedTile = false;

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

        Unit.OnAnyUnitDead += OnAnyUnitDead;
        SceneManager.sceneLoaded += OnSceneLoaded;
    }

    private void OnDestroy()
    {
        Unit.OnAnyUnitDead -= OnAnyUnitDead;
        SceneManager.sceneLoaded -= OnSceneLoaded;
    }

    private void OnSceneLoaded(Scene s, LoadSceneMode m)
    {
        StartCoroutine(Co_ClearAfterSceneLoad());
    }

    private IEnumerator Co_ClearAfterSceneLoad()
    {
        yield return null;
        PurgeDeadAndNullWatchers();
        ClearAllWatchers();
        GridSystemVisual.Instance.ClearAllPersistentOverwatch();
```

```
    }

    public float GetEnemyOverwatchVisibilityDuration()
    {
        return enemyOverwatchVisibilityDuration;
    }

    public void UnitTurnEndStatus(IEnumerable<Unit> teamUnits)
    {
        PurgeDeadAndNullWatchers();

        foreach (var unit in teamUnits)
        {
            if (!unit) continue;

            var vision = unit.GetComponent<UnitVision>();
            if (vision == null || !vision.IsInitialized) continue;

            int ap = unit.GetActionPoints();
            float angle = vision.VisionPenaltyWhenUsingAP(ap);

            if (unit.TryGetComponent<OverwatchAction>(out var ow) && ow.IsOverwatch())
            {
                vision.UpdateVisionNow();

                var settings = ow.GetOverwatchSettings();
                Vector3 facing = unit.transform.forward;
                float coneAngle = settings.coneAngleDeg;

                if (unit.TryGetComponent<UnitStatusController>(out var status) &&
                    status.TryGet<OverwatchPayload>(UnitStatusType.Overwatch, out var payload))
                {
                    facing = payload.facingWorld;
                    coneAngle = payload.coneAngleDeg;
                    if (facing.sqrMagnitude < 1e-4f) facing = unit.transform.forward;

                    if (Mathf.Abs(payload.coneAngleDeg - settings.coneAngleDeg) > 0.1f ||
                        payload.rangeTiles != settings.rangeTiles)
                    {
                        payload.coneAngleDeg = settings.coneAngleDeg;
                        payload.rangeTiles = settings.rangeTiles;
                        status.AddOrUpdate(UnitStatusType.Overwatch, payload);
                    }
                }
                else
                {
                    var dir = ow.TargetWorld - unit.transform.position; dir.y = 0f;
                    if (dir.sqrMagnitude > 1e-4f) facing = dir.normalized;
                }

                var coneTiles = vision.GetConeVisibleTiles(facing, coneAngle);
                if (coneTiles != null && TeamVisionService.Instance != null)
```

```
            {
                int teamId = unit.GetTeamID();
                int unitKey = vision.GetInstanceID();
                TeamVisionService.Instance.ReplaceUnitVision(teamId, unitKey, new HashSet<GridPosition>(coneTiles));
            }

            continue;
        }

        if (angle >= 359.5f)
        {
            vision.UpdateVisionNow();
            continue;
        }

        Vector3 facingNormal = unit.transform.forward;
        vision.ApplyAndPublishDirectionalVision(facingNormal, angle);
    }

    SetOverWatch(teamUnits);
}

public void UnitTurnStartStatus(IEnumerable<Unit> teamUnits)
{
    PurgeDeadAndNullWatchers();
    if (teamUnits == null) return;

    ForceCleanStateForTurnStart(teamUnits);

    foreach (var u in teamUnits)
    {
        if (!u) continue;

        if (u.TryGetComponent<UnitVision>(out var vision) && vision.IsInitialized)
        {
            vision.UpdateVisionNow();
        }
    }

    RemoveOverWatchSafe(teamUnits);
}

private void RemoveOverWatchSafe(IEnumerable<Unit> teamUnits)
{
    if (teamUnits == null) return;

    foreach (var unit in teamUnits)
    {
        if (!unit) continue;

        if (unit.TryGetComponent<UnitStatusController>(out var status) &&
            status.Has(UnitStatusType.Overwatch))
```

```
            {
                status.Remove(UnitStatusType.Overwatch);
            }

            GridSystemVisual.Instance.RemovePersistentOverwatch(unit);

            if (unit.TryGetComponent<OverwatchAction>(out var action))
            {
                action.CancelOverwatchIntent();
            }

            RemoveWatcher(unit);
        }
    }

    public void AddWatcher(Unit unit)
    {
        int teamId = unit.GetTeamID();

        if (!overwatchByTeam.TryGetValue(teamId, out var set))
        {
            set = new HashSet<Unit>();
            overwatchByTeam[teamId] = set;
        }

        set.Add(unit);
    }

    public void RemoveWatcher(Unit unit)
    {
        int teamId = unit.GetTeamID();

        if (overwatchByTeam.TryGetValue(teamId, out var set))
        {
            set.Remove(unit);
            if (set.Count == 0) overwatchByTeam.Remove(teamId);
            _nextReactAt.Remove(unit);
        }
    }

    private void SetOverWatch(IEnumerable<Unit> teamUnits)
    {
        if (!NetworkServer.active && !NetworkSync.IsOffline)
        {
            Debug.LogWarning("[OW-StatusCoord] SetOverWatch called on non-server in online game - this should not happen!");
            return;
        }

        foreach (var unit in teamUnits)
        {
            if (!unit) continue;
```

```
            var action = unit.GetComponent<OverwatchAction>();
            if (action != null && action.IsOverwatch())
            {
                var status = unit.GetComponent<UnitStatusController>();
                if (status == null)
                {
                    Debug.LogWarning($"[OW-StatusCoord] {unit.name} has no UnitStatusController!");
                    continue;
                }

                if (!status.Has(UnitStatusType.Overwatch))
                {
                    var settings = action.GetOverwatchSettings();
                    var setup = new OverwatchPayload
                    {
                        facingWorld = unit.transform.forward,
                        coneAngleDeg = settings.coneAngleDeg,
                        rangeTiles = settings.rangeTiles
                    };
                    status.AddOrUpdate(UnitStatusType.Overwatch, setup);
                }

                AddWatcher(unit);
            }
        }
    }

    public IEnumerable<Unit> GetWatchers(int teamId)
    {
        return overwatchByTeam.TryGetValue(teamId, out var set) ? set : System.Array.Empty<Unit>();
    }

    private bool CanReactNow(Unit w) => !_nextReactAt.TryGetValue(w, out var t) || Time.time >= t;

    private void MarkReactedNow(Unit w, float cooldown)
    {
        if (w) _nextReactAt[w] = Time.time + cooldown;
    }

    public void CheckOverwatchStep(Unit mover, GridPosition newGridPos)
    {
        int enemyTeamId = (mover.GetTeamID() == 0) ? 1 : 0;
        var watchers = GetWatchers(enemyTeamId);

        foreach (var watcher in watchers)
        {
            if (!watcher || watcher.IsDead() || watcher.IsDying()) continue;

            if (!watcher.TryGetComponent<UnitVision>(out var vision) || !vision.IsInitialized) continue;

            Vector3 facingWorld;
            float coneDeg;
```

```
        if (watcher.TryGetComponent<UnitStatusController>(out var status) &&
            status.TryGet<OverwatchPayload>(UnitStatusType.Overwatch, out var payload))
        {
            facingWorld = payload.facingWorld;
            coneDeg = payload.coneAngleDeg;
        }
        else
        {
            facingWorld = OverwatchHelpers.NormalizeFacing(watcher.transform.forward);
            coneDeg = 80f;
        }

        var personal = vision.GetUnitVisionGrids();
        if (personal == null || personal.Count == 0)
        {
            vision.UpdateVisionNow();
            personal = vision.GetUnitVisionGrids();
        }
        if (personal == null || !personal.Contains(newGridPos))
            continue;

        if (!vision.IsTileInCone(facingWorld, coneDeg, newGridPos))
            continue;

        if (!CanReactNow(watcher))
        {
            continue;
        }

        var owAction = watcher.GetComponent<OverwatchAction>();
        var settings = owAction != null ? owAction.GetOverwatchSettings() : OverwatchShootingSettings.Default;

        MarkReactedNow(watcher, settings.reactionCooldownSeconds);

        var target = LevelGrid.Instance.GetUnitAtGridPosition(newGridPos);

        StartCoroutine(Co_TriggerOverwatchWithJitter(watcher, target, newGridPos, settings.reactionJitterMaxSeconds));

        if (onlyOneOverwatchAttackPerMovedTile) break;
    }
}

private IEnumerator Co_TriggerOverwatchWithJitter(Unit watcher, Unit target, GridPosition posAtTrigger, float jitterMax)
{
    float delay = UnityEngine.Random.Range(0f, jitterMax);
    if (delay > 0f) yield return new WaitForSeconds(delay);

    if (!NetworkSync.IsOffline && !Mirror.NetworkServer.active) yield break;
    if (!watcher || watcher.IsDead() || watcher.IsDying()) yield break;
    if (!target  || target.IsDead()  || target.IsDying())  yield break;
    if (watcher.GetReactionPoints() <= 0) yield break;
```

```
        watcher.SpendReactionPoints();
        NetworkSync.TriggerOverwatchShot(watcher, target, posAtTrigger);
    }

    private void OnAnyUnitDead(object sender, EventArgs e)
    {
        var dead = sender as Unit;

        if (dead)
        {
            GridSystemVisual.Instance.RemovePersistentOverwatch(dead);
            if (dead.TryGetComponent<OverwatchAction>(out var ow))
                ow.CancelOverwatchIntent();
        }

        PurgeDeadAndNullWatchers(dead);
        _nextReactAt.Remove(dead);
    }

    private void PurgeDeadAndNullWatchers(Unit maybeDead = null)
    {
        if (overwatchByTeam.Count == 0) return;

        var teams = new List<int>(overwatchByTeam.Keys);
        foreach (var team in teams)
        {
            if (!overwatchByTeam.TryGetValue(team, out var set) || set == null)
            {
                overwatchByTeam.Remove(team);
                continue;
            }

            set.RemoveWhere(unit => !unit || ReferenceEquals(unit, maybeDead) || unit.IsDead() || unit.IsDying());

            if (set.Count == 0)
                overwatchByTeam.Remove(team);
        }
    }

    public void ClearAllWatchers()
    {
        overwatchByTeam.Clear();
        _nextReactAt.Clear();
    }

    public void ForceCleanStateForTurnStart(IEnumerable<Unit> teamUnits)
    {
        if (teamUnits == null) return;

        // 1. Pyyhi kaikki vanhat overwatch-visualisoinnit
        GridSystemVisual.Instance.ClearAllPersistentOverwatch();
```

```
        // 2. Pyyhi kaikki vanhat grid-merkinnät
        GridSystemVisual.Instance.HideAllGridPositions();

        // 3. Pakota jokaisen yksikön vision päivitys täyteen 360°
        foreach (var unit in teamUnits)
        {
            if (!unit || unit.IsDead() || unit.IsDying()) continue;

            // Varmista että UnitVision on initialisoitu
            if (unit.TryGetComponent<UnitVision>(out var vision) && vision.IsInitialized)
            {
                // Pakota täysi vision päivitys (360°, ei kartiota)
                vision.UpdateVisionNow();

                // Varmista että TeamVisionService saa päivityksen
                int teamId = unit.GetTeamID();
                int unitKey = vision.GetInstanceID();
                var visionTiles = vision.GetUnitVisionGrids();

                if (visionTiles != null && TeamVisionService.Instance != null)
                {
                    TeamVisionService.Instance.ReplaceUnitVision(teamId, unitKey, visionTiles);
                }
            }
        }

        // 4. Pakota grid-visualisoinnin päivitys
        if (GridSystemVisual.Instance != null)
        {
            GridSystemVisual.Instance.UpdateGridVisuals();
        }
    }
}
```

Assets/scripts/Units/UnitStatus/UnitStatusController.cs

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

public enum UnitStatusType { Overwatch /*, Stunned, Wounded, Suppressed, ...*/ }

public interface IStatusPayload { }

// Overwatchin parametrit statuksena
public struct OverwatchPayload : IStatusPayload {
    public Vector3 facingWorld;   // tai esim. yaw-aste
    public float coneAngleDeg;    // esim. 80
    public int   rangeTiles;      // esim. 8
}

public class UnitStatusController : MonoBehaviour {

    private readonly Dictionary<UnitStatusType, object> _map = new();

    public event Action<UnitStatusType> OnAdded;
    public event Action<UnitStatusType> OnRemoved;
    public event Action<UnitStatusType> OnChanged;

    public bool Has(UnitStatusType statusType) => _map.ContainsKey(statusType);

    public bool TryGet<T>(UnitStatusType statusType, out T setup) where T: struct, IStatusPayload {
        if (_map.TryGetValue(statusType, out var obj) && obj is T p) { setup = p; return true; }
        setup = default; return false;
    }

    public void AddOrUpdate<T>(UnitStatusType statusType, T setup) where T: struct, IStatusPayload {
        bool existed = _map.ContainsKey(statusType);
        _map[statusType] = setup;
        if (existed) OnChanged?.Invoke(statusType); else OnAdded?.Invoke(statusType);
    }

    public void Remove(UnitStatusType statusType) {
        if (_map.Remove(statusType)) OnRemoved?.Invoke(statusType);
    }
}
```

Assets/scripts/Units/Vision/TeamVisionService.cs

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

/// <summary>
/// Global, scene-persistent service that maintains each team's aggregate
/// line-of-sight/visibility as the union of all units' vision sets.
/// </summary>
/// <remarks>
/// - Lives across scene loads via <see cref="Object.DontDestroyOnLoad(UnityEngine.Object)"/>.
/// - Visibility is tracked per <c>teamId</c>.
/// - Each unit contributes a set of <see cref="GridPosition"/> values.
/// - Internally, overlapping vision is handled with reference counting,
///   so a tile stays visible while at least one unit still sees it.
/// - Raises <see cref="OnTeamVisionChanged"/> whenever a team's aggregate
///   vision changes.
/// - Intended for use from Unity's main thread only.
/// </remarks>
public class TeamVisionService : MonoBehaviour
{
    /// <summary>
    /// Singleton instance of the service.
    /// </summary>
    public static TeamVisionService Instance { get; private set; }

    /// <summary>
    /// Fired when a team's aggregate vision changes.
    /// The argument is the affected <c>teamId</c>.
    /// </summary>
    public event Action<int> OnTeamVisionChanged;

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

    /// <summary>
    /// Per-team accumulators of visible tiles.
    /// </summary>
    private readonly Dictionary<int, VisionAccumulator> _teams = new();

    /// <summary>
    /// Returns a snapshot (copy) of all tiles currently visible to the given team.
    /// </summary>
```

```
    /// <param name="teamId">Team identifier.</param>
    /// <returns>Read-only collection of visible <see cref="GridPosition"/> tiles.</returns>
    public IReadOnlyCollection<GridPosition> GetVisibleTilesSnapshot(int teamId)
    {
        var snapshot = GetAcc(teamId).GetVisibleSnapshot();
        return snapshot;
    }

    /// <summary>
    /// Notifies listeners that a team's vision changed.
    /// </summary>
    /// <param name="teamId">Team identifier.</param>
    private void NotifyTeamChanged(int teamId)
    {
        OnTeamVisionChanged?.Invoke(teamId);
    }

    /// <summary>
    /// Gets (or lazily creates) the accumulator for the given team.
    /// </summary>
    /// <param name="teamId">Team identifier.</param>
    /// <returns>Accumulator for that team.</returns>
    private VisionAccumulator GetAcc(int teamId)
    {
        if (!_teams.TryGetValue(teamId, out var acc))
        {
            _teams[teamId] = acc = new VisionAccumulator();
        }
        return acc;
    }

    /// <summary>
    /// Replaces a unit's visible-tile set for a team, updating the team's aggregate vision.
    /// </summary>
    /// <param name="teamId">Team identifier.</param>
    /// <param name="unitKey">
    /// Stable key identifying the unit (e.g., instance ID, net ID, or other unique handle).
    /// </param>
    /// <param name="newSet">The unit's current visible tiles.</param>
    public void ReplaceUnitVision(int teamId, int unitKey, HashSet<GridPosition> newSet)
    {
        GetAcc(teamId).ReplaceUnitSet(unitKey, newSet);
        NotifyTeamChanged(teamId);
    }

    /// <summary>
    /// Removes a unit's contribution to a team's vision (e.g., on despawn or death).
    /// </summary>
    /// <param name="teamId">Team identifier.</param>
    /// <param name="unitKey">Stable key identifying the unit.</param>
    public void RemoveUnitVision(int teamId, int unitKey)
    {
```

```
        GetAcc(teamId).RemoveUnitSet(unitKey);
        NotifyTeamChanged(teamId);
    }

    /// <summary>
    /// Clears all visibility data for a team.
    /// </summary>
    /// <param name="teamId">Team identifier.</param>
    public void ClearTeamVision(int teamId)
    {
        if (_teams.TryGetValue(teamId, out var acc))
        {
            acc.Clear();
            NotifyTeamChanged(teamId);
        }
    }

    /// <summary>
    /// Returns whether the specified tile is currently visible to the team.
    /// </summary>
    /// <param name="teamId">Team identifier.</param>
    /// <param name="gp">Grid position to query.</param>
    /// <returns><c>true</c> if visible; otherwise <c>false</c>.</returns>
    public bool IsVisibleToTeam(int teamId, GridPosition gp)
        => GetAcc(teamId).IsVisible(gp);

    /// <summary>
    /// Internal helper that aggregates unit vision for a single team using
    /// reference counting per tile.
    /// </summary>
    private class VisionAccumulator
    {
        /// <summary>
        /// Per-unit visible-tile sets.
        /// </summary>
        private readonly Dictionary<int, HashSet<GridPosition>> _unitSets = new();

        /// <summary>
        /// Reference counts per tile across all units in the team.
        /// </summary>
        private readonly Dictionary<GridPosition, int> _counts = new();

        /// <summary>
        /// Replaces the stored set for one unit and updates per-tile reference counts.
        /// </summary>
        /// <param name="unitKey">Unit identifier.</param>
        /// <param name="newSet">The unit's current visible tiles.</param>
        public void ReplaceUnitSet(int unitKey, HashSet<GridPosition> newSet)
        {
            if (_unitSets.TryGetValue(unitKey, out var oldSet))
            {
                foreach (var gp in oldSet)
```

```
            {
                if (_counts.TryGetValue(gp, out int c))
                {
                    c--; if (c <= 0) _counts.Remove(gp); else _counts[gp] = c;
                }
            }
        }

        _unitSets[unitKey] = newSet;

        foreach (var gp in newSet)
        {
            _counts.TryGetValue(gp, out int c);
            _counts[gp] = c + 1;
        }
    }

    /// <summary>
    /// Removes a unit's contribution entirely and updates reference counts.
    /// </summary>
    /// <param name="unitKey">Unit identifier.</param>
    public void RemoveUnitSet(int unitKey)
    {
        if (!_unitSets.TryGetValue(unitKey, out var oldSet)) return;
        foreach (var gp in oldSet)
        {
            if (_counts.TryGetValue(gp, out int c))
            {
                c--; if (c <= 0) _counts.Remove(gp); else _counts[gp] = c;
            }
        }
        _unitSets.Remove(unitKey);
    }

    /// <summary>
    /// Clears all unit data and all reference counts for the team.
    /// </summary>
    public void Clear()
    {
        _unitSets.Clear();
        _counts.Clear();
    }

    /// <summary>
    /// Returns a snapshot (copy) of all tiles that are currently visible
    /// to at least one unit in the team.
    /// </summary>
    public IReadOnlyCollection<GridPosition> GetVisibleSnapshot()
    {
        return new List<GridPosition>(_counts.Keys);
    }
```

```
        /// <summary>
        /// Returns whether a tile has a non-zero reference count (i.e., is visible).
        /// </summary>
        /// <param name="gp">Grid position to query.</param>
        public bool IsVisible(GridPosition gp) => _counts.ContainsKey(gp);
    }

    public void RebuildTeamVisionLocal(int teamId, bool endPhase)
    {
        var list = UnitManager.Instance?.GetAllUnitList();
        if (list == null) return;

        foreach (var unit in list)
        {
            if (!unit) continue;
            if (unit.GetTeamID() != teamId) continue;
            if (unit.IsDead() || unit.IsDying()) continue;

            var vision = unit.GetComponent<UnitVision>();
            if (vision == null || !vision.IsInitialized) continue;

            // Aina päivitä tuore 360° cache ensin
            vision.UpdateVisionNow();
            int actionpoints = unit.GetActionPoints();
            Vector3 facing = unit.transform.forward;
            float angle = endPhase ? vision.VisionPenaltyWhenUsingAP(actionpoints) : 360f;

            if (endPhase && unit.TryGetComponent<OverwatchAction>(out var ow) && ow.IsOverwatch())
            {
                angle = vision.VisionPenaltyWhenUsingAP(0);
                var dir = ow.TargetWorld - unit.transform.position; dir.y = 0f;
                if (dir.sqrMagnitude > 1e-4f) facing = dir.normalized;

            }

            vision.ApplyAndPublishDirectionalVision(facing, angle);
        }
    }
}
```

Assets/scripts/Units/Vision/UnitVision.cs

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

[DisallowMultipleComponent]
public class UnitVision : MonoBehaviour
{

    [Header("Overwatch cone")]
    [SerializeField, Range(0f, 2f)]
    private float coneOriginBackshiftTiles = 1f; // 1 = yksi ruutu taaksepäin

    [Header("Config")]
    public UnitArchetype visionSkill;
    public int teamId = 0;

    private Unit _unit;
    private Transform _tr;
    private HashSet<GridPosition> _lastVisibleTiles = new();
    private int _unitKey;
    private bool _initialized = false;
    private int _currentTeamId = 0;
    private HealthSystem _healthSystem;

    void Awake()
    {
        _tr = transform;
        _unit = GetComponent<Unit>();
        _healthSystem = GetComponent<HealthSystem>();
        _unitKey = GetInstanceID();
    }

    void OnEnable()
    {
        if (_healthSystem != null)
            _healthSystem.OnDying += HandleUnitDying;
    }

    void Start()
    {
        if (!_initialized)
            StartCoroutine(Co_AutoInitLocal());
    }

    void OnDisable()
    {
        if (_healthSystem != null)
            _healthSystem.OnDying -= HandleUnitDying;

        CleanupVision();
```

```
    }

    private void HandleUnitDying(object sender, System.EventArgs e)
    {
        CleanupVision();

        GridSystemVisual.Instance.RemovePersistentOverwatch(_unit);
        StatusCoordinator.Instance.RemoveWatcher(_unit);
    }

    private void CleanupVision()
    {
        if (TeamVisionService.Instance != null && _initialized)
        {
            TeamVisionService.Instance.RemoveUnitVision(teamId, _unitKey);
        }
        _lastVisibleTiles.Clear();
    }

    private System.Collections.IEnumerator Co_AutoInitLocal()
    {
        yield return null;

        if (_initialized) yield break;
        if (NetMode.IsDedicatedServer) yield break;

        for (int attempt = 0; attempt < 30; attempt++)
        {
            int? team = _unit.GetTeamID();

            if (team != null)
            {
                if (_unit == null) _unit = GetComponent<Unit>();
                InitializeVision(team.Value, _unit ? _unit.archetype : visionSkill);
                yield break;
            }

            yield return null;
        }
    }

    public void InitializeVision(int setTeamId, UnitArchetype archetype)
    {
        if (_initialized && setTeamId != _currentTeamId && TeamVisionService.Instance != null)
            TeamVisionService.Instance.RemoveUnitVision(_currentTeamId, _unitKey);

        _currentTeamId = setTeamId;
        teamId = setTeamId;
        if (archetype != null) visionSkill = archetype;

        _initialized = true;
```

```
        StartCoroutine(Co_DeferredFirstVision());
    }

    private System.Collections.IEnumerator Co_DeferredFirstVision()
    {
        yield return null;
        UpdateVisionNow();
    }

    public void NotifyMoved()
    {
        UpdateVisionNow();
    }

    public bool IsInitialized => _initialized;

    public void UpdateVisionNow()
    {
        if (_unit != null && (_unit.IsDying() || _unit.IsDead())) return;
        if (_healthSystem != null && (_healthSystem.IsDying() || _healthSystem.IsDead())) return;

        bool publish = ShouldPublishVisionLocally();

        if (!_initialized)
        {
            Debug.LogWarning($"[UnitVision UPDATE SKIP] {name} - not initialized");
            return;
        }
        if (visionSkill == null)
        {
            Debug.LogWarning($"[UnitVision UPDATE SKIP] {name} - no visionSkill");
            return;
        }

        var levelGrid = LevelGrid.Instance;
        var loSConfig = LoSConfig.Instance;
        if (levelGrid == null || loSConfig == null)
        {
            Debug.LogWarning($"[UnitVision UPDATE SKIP] {name} - missing services: LG={levelGrid != null}, CFG={loSConfig != null}");
            return;
        }

        var wp = _tr != null ? _tr.position : transform.position;
        var origin = levelGrid.GetGridPosition(wp);
        if (_unit != null)
        {
            var uf = _unit.GetGridPosition();
            origin = new GridPosition(origin.x, origin.z, uf.floor);
        }

        HashSet<GridPosition> vis;
        if (visionSkill.useHeightAware)
```

```
        {
            vis = RaycastVisibility.ComputeVisibleTilesRaycastHeightAware(
                origin, visionSkill.visionRange,
                loSConfig.losBlockersMask, loSConfig.eyeHeight, loSConfig.samplesPerCell, loSConfig.insetWU,
                ignoreRoot: _tr
            );
        }
        else
        {
            vis = RaycastVisibility.ComputeVisibleTilesRaycast(
                origin, visionSkill.visionRange,
                loSConfig.losBlockersMask, loSConfig.eyeHeight, loSConfig.samplesPerCell, loSConfig.insetWU
            );
        }

        if (vis != null) _lastVisibleTiles = vis;

        if (publish)
        {
            var teamVision = TeamVisionService.Instance;
            if (teamVision != null)
                teamVision.ReplaceUnitVision(teamId, _unitKey, _lastVisibleTiles);
        }
    }

    private bool ShouldPublishVisionLocally()
    {
        if (NetworkSync.IsOffline) return true;

        if (!NetworkSync.IsClient) return false;

        if (GameModeManager.SelectedMode == GameMode.Versus)
        {
            if (!PlayerLocalTurnGate.LocalPlayerTurn) return false;

            var ni = NetworkSync.FindIdentity(this.GetActorId());
            return NetworkSync.IsOwnedHere(ni);
        }

        {
            var ni = NetworkSync.FindIdentity(this.GetActorId());
            return NetworkSync.IsOwnedHere(ni);
        }
    }

    public HashSet<GridPosition> GetUnitVisionGrids()
    {
        return _lastVisibleTiles;
    }

    public void ShowUnitPersonalVision()
    {
```

```
        var tiles = GetUnitVisionGrids();
        if (tiles == null) return;

        GridSystemVisual.Instance.ShowGridPositionList(
            new List<GridPosition>(tiles),
            GridVisualType.UnitPersonalVision
        );
    }

    public void ShowUnitOverWatchVision(Vector3 facingWorld, float coneAngleDeg)
    {
        var tiles = GetConeVisibleTiles(facingWorld, coneAngleDeg);
        if (tiles == null || tiles.Count == 0) return;

        GridSystemVisual.Instance.AddPersistentOverwatch(_unit, tiles);
    }

    private static float EstimateCellSizeWU(LevelGrid lg, GridPosition gp)
    {
        var a = lg.GetWorldPosition(gp);
        var b = lg.GetWorldPosition(new GridPosition(gp.x + 1, gp.z, gp.floor));
        float dx = Mathf.Abs(b.x - a.x);
        if (dx > 1e-4f) return dx;

        // fallback Z-suunnasta
        var c = lg.GetWorldPosition(new GridPosition(gp.x, gp.z + 1, gp.floor));
        return Mathf.Abs(c.z - a.z);
    }

    public List<GridPosition> GetConeVisibleTiles(Vector3 facingWorld, float coneAngleDeg)
    {
        var visible = GetUnitVisionGrids();
        var levelGrid = LevelGrid.Instance;
        var res = new List<GridPosition>();
        if (visible == null || visible.Count == 0 || levelGrid == null) return res;

        // facing 2D
        Vector2 f = new Vector2(facingWorld.x, facingWorld.z);
        if (f.sqrMagnitude < 1e-6f) f = new Vector2(transform.forward.x, transform.forward.z);
        f.Normalize();

        float cosHalf = Mathf.Cos(0.5f * coneAngleDeg * Mathf.Deg2Rad);

        var myGridPos = _unit.GetGridPosition();
        int myFloor = myGridPos.floor;

        // --- UUSI: siirrä alkuperää yhden ruudun taaksepäin facing-suunnassa ---
        float cell = EstimateCellSizeWU(levelGrid, myGridPos);
        Vector3 myGridCenter = levelGrid.GetWorldPosition(myGridPos);
        Vector3 shiftedOrigin = myGridCenter - new Vector3(f.x, 0f, f.y) * (coneOriginBackshiftTiles * cell);

        foreach (var gp in visible)
```

```
        {
            if (gp.floor != myFloor) continue;

            // Oma ruutu voi edelleen olla poissa (ohitetaan se kuten ennenkin)
            if (gp.x == myGridPos.x && gp.z == myGridPos.z)
                continue;

            Vector3 wp = levelGrid.GetWorldPosition(gp);

            // --- UUSI: mittaa vektori siirretystä alkuperästä ---
            Vector2 to = new Vector2(wp.x - shiftedOrigin.x, wp.z - shiftedOrigin.z);
            float len2 = to.sqrMagnitude;
            if (len2 < 1e-6f) continue;

            Vector2 toN = to / Mathf.Sqrt(len2);
            float dot = Vector2.Dot(f, toN);
            if (dot >= cosHalf)
                res.Add(gp);
        }

        return res;
    }

    public float VisionPenaltyWhenUsingAP(int apLeft, float VisionAngleDegPenalty = 80f)
    {
        if (apLeft >= 3) return 360f;
        if (apLeft == 2) return 360f - VisionAngleDegPenalty;
        if (apLeft == 1) return 180f;
        return VisionAngleDegPenalty;
    }

    public bool IsTileInCone(Vector3 facingWorld, float coneAngleDeg, GridPosition gp)
    {
        var levelGrid = LevelGrid.Instance;
        if (levelGrid == null || _unit == null) return false;

        var myGridPos = _unit.GetGridPosition();
        int myFloor = myGridPos.floor;
        if (gp.floor != myFloor) return false;

        // Oma ruutu ei ole “kohderuutu” -> pidetään poissa
        if (gp.x == myGridPos.x && gp.z == myGridPos.z)
            return false;

        Vector2 f = new Vector2(facingWorld.x, facingWorld.z);
        if (f.sqrMagnitude < 1e-6f) f = new Vector2(transform.forward.x, transform.forward.z);
        f.Normalize();

        float cosHalf = Mathf.Cos(0.5f * coneAngleDeg * Mathf.Deg2Rad);

        // --- UUSI: siirretty alkuperä ---
        float cell = EstimateCellSizeWU(levelGrid, myGridPos);
```

```
        Vector3 myGridCenter = levelGrid.GetWorldPosition(myGridPos);
        Vector3 shiftedOrigin = myGridCenter - new Vector3(f.x, 0f, f.y) * (coneOriginBackshiftTiles * cell);

        Vector3 wp = levelGrid.GetWorldPosition(gp);
        Vector2 to = new Vector2(wp.x - shiftedOrigin.x, wp.z - shiftedOrigin.z);
        float len2 = to.sqrMagnitude;
        if (len2 < 1e-6f) return false;

        Vector2 toN = to / Mathf.Sqrt(len2);
        return Vector2.Dot(f, toN) >= cosHalf;
    }

    public void ApplyAndPublishDirectionalVision(Vector3 facingWorld, float coneAngleDeg)
    {
        var tiles = GetConeVisibleTiles(facingWorld, coneAngleDeg);
        if (tiles == null) return;

        _lastVisibleTiles = new HashSet<GridPosition>(tiles);
        TeamVisionService.Instance.ReplaceUnitVision(teamId, _unitKey, _lastVisibleTiles);
    }
}
```

Assets/scripts/Utilities/OverwatchHelpers.cs

```
using UnityEngine;

public static class OverwatchHelpers
{
    public static Vector3 NormalizeFacing(Vector3 facing)
    {
        facing.y = 0f;

        if (facing.sqrMagnitude < 1e-6f)
        {
            return Vector3.forward;
        }

        return facing.normalized;
    }
}
```

Assets/scripts/Utilities/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/Utilities/SircleCalculator.cs

```
using UnityEngine;

public static class SircleCalculator
{

   public static int Sircle(int x, int z)
    {
        int ax = Mathf.Abs(x);
        int az = Mathf.Abs(z);
        int diag = Mathf.Min(ax, az);
        int straight = Mathf.Abs(ax - az);
        int cost = 14 * diag + 10 * straight;

        return cost;
    }
}
```

Assets/scripts/Visibility/LocalVisibility.cs

```
// LocalVisibility.cs
using UnityEngine;

public class LocalVisibility : MonoBehaviour
{
    Renderer[] _renderers;
    Canvas[] _canvases;
    public bool IsVisible { get; private set; } = true;

    void Awake()
    {
        _renderers = GetComponentsInChildren<Renderer>(true);
        _canvases  = GetComponentsInChildren<Canvas>(true);
    }

    public void Apply(bool visible)
    {
        if (IsVisible == visible) return; // ei turhaa työtä
        IsVisible = visible;

        // Toggle vain kun oikeasti muuttuu
        for (int i = 0; i < _renderers.Length; i++) _renderers[i].enabled = visible;
        for (int i = 0; i < _canvases.Length;  i++) _canvases[i].enabled  = visible;
    }
}
```

Assets/scripts/Visibility/LoSConfing.cs

```
using UnityEngine;

[CreateAssetMenu(fileName = "LoSConfig", menuName = "Config/LoS Config")]
public class LoSConfig : ScriptableObject
{
    public LayerMask losBlockersMask;      // vain korkeat seinät
    public float     eyeHeight      = 1.6f;
    [Range(1,5)]
    public int       samplesPerCell = 1;   // 1=nopea, 5=kulmiin jämäkämpi
    [Range(0f, 0.5f)]
    public float     insetWU        = 0.30f;

    private static LoSConfig _instance;
    public static LoSConfig Instance {
        get {
            if (_instance == null)
                _instance = Resources.Load<LoSConfig>("LoSConfig");
#if UNITY_EDITOR
            if (_instance == null)
                Debug.LogError("LoSConfig asset puuttuu: luo Resources/LoSConfig.asset (Create > Config > LoS Config).");
#endif
            return _instance;
        }
    }
}
```

Assets/scripts/Visibility/RaycastVisibility.cs

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


public interface IDynamicLoSBlocker
{
    /// <summary>
    /// Palauta true jos tämä objekti blokkaa LoS:n silmäkorkeudella (worldY).
    /// </summary>
    bool IsBlockingAtHeight(float heightWorldY);
}


public static class RaycastVisibility
{
    private const float EPS = 0.01f;
    private static readonly RaycastHit[] _hitBuf1 = new RaycastHit[1];

    public enum LoSAcceptance
    {
        Any,                // nykyinen “yksi riittää” (ei suositella)
        CenterOnly,         // vain keskisäde ratkaisee (reilu & tiukka)
        CenterAndAny,       // keskisäde + väh. yksi muu
        Majority,           // väh. 3/5 selvä (offsetit “pehmentää”)
        All                 // kaikki 5 läpi (tosi tiukka)
    }

    /// <summary>
    /// Yksinkertainen LoS: vain losBlockersMask (korkeat seinät tms.) blokkaa.
    /// samplesPerCell: 1 = nopea (keskipiste), 5 = “risti” (kulmasuoja pienenee).
    /// insetWU siirtää kohdepisteitä hieman ruudun sisään, jotta reunaräpsyt vähenee.
    /// </summary>
    public static bool HasLineOfSightRaycast(
        GridPosition from, GridPosition to,
        LayerMask losBlockersMask,
        float eyeHeight = 1.6f,
        int samplesPerCell = 1,
        float insetWU = 0.30f)
    {
        if (from.floor != to.floor) return false;

        var lg = LevelGrid.Instance;
        if (lg == null) return false;

        Vector3 a = lg.GetWorldPosition(from) + Vector3.up * eyeHeight;
        Vector3 center = lg.GetWorldPosition(to) + Vector3.up * eyeHeight;

        if (samplesPerCell <= 1)
            return RayClear(a, center, losBlockersMask);
```

```
        // 5-pisteen “risti”: center ±right ±forward
        Vector3 dir = (center - a).normalized;
        Vector3 right = Vector3.Cross(Vector3.up, dir).normalized;
        Vector3 forward = Vector3.Cross(right, Vector3.up).normalized;

        Vector3[] targets = new[]
        {
            center,
            center + right   * insetWU,
            center - right   * insetWU,
            center + forward * insetWU,
            center - forward * insetWU,
        };

        // Yksi “onnistunut” säde riittää näyttämään ruudun näkyvänä.
        // Jos haluat “täysin näkyvä ruutu” -säännön, vaadi että kaikki läpäisee.
        foreach (var t in targets)
            if (RayClear(a, t, losBlockersMask)) return true;

        return false;
    }

    public static HashSet<GridPosition> ComputeVisibleTilesRaycast(
        GridPosition origin,
        int maxRange,
        LayerMask losBlockersMask,
        float eyeHeight = 1.6f,
        int samplesPerCell = 1,
        float insetWU = 0.30f)
    {
        var set = new HashSet<GridPosition>();
        var lg = LevelGrid.Instance;
        if (lg == null) return set;

        for (int dx = -maxRange; dx <= maxRange; dx++)
            for (int dz = -maxRange; dz <= maxRange; dz++)
            {
                if (dx == 0 && dz == 0) continue;
                var gp = origin + new GridPosition(dx, dz, 0);
                if (!lg.IsValidGridPosition(gp)) continue;

                int cost = SircleCalculator.Sircle(dx, dz);
                if (cost > 10 * maxRange) continue;

                if (HasLineOfSightRaycast(origin, gp, losBlockersMask, eyeHeight, samplesPerCell, insetWU))
                    set.Add(gp);
            }
        return set;
    }

    private static bool RayClear(Vector3 a, Vector3 b, LayerMask mask)
```

```
    {
        Vector3 d = b - a;
        float L = d.magnitude;
        if (L <= EPS) return true;
        int hits = Physics.RaycastNonAlloc(a, d / L, _hitBuf1, L - EPS, mask, QueryTriggerInteraction.Ignore);
        return hits == 0;
    }

    public static bool HasLineOfSightRaycastHeightAware(
        GridPosition from, GridPosition to,
        LayerMask losMask,
        float eyeHeight = 1.6f,
        int samplesPerCell = 1,
        float insetWU = 0.30f,
        Transform ignoreRoot = null,
        LoSAcceptance acceptance = LoSAcceptance.CenterOnly
    )
    {
        if (from.floor != to.floor) return false;

        var lg = LevelGrid.Instance;
        if (lg == null) return false;

        Vector3 a = lg.GetWorldPosition(from) + Vector3.up * eyeHeight;
        Vector3 center = lg.GetWorldPosition(to) + Vector3.up * eyeHeight;

        if (samplesPerCell <= 1)
            return RayClearHeightAware(a, center, losMask, ignoreRoot);

        Vector3 dir = (center - a).normalized;
        Vector3 right   = Vector3.Cross(Vector3.up, dir).normalized;
        Vector3 forward = Vector3.Cross(right,   Vector3.up).normalized;

        Vector3[] targets = new[]
        {
            center,
            center + right   * insetWU,
            center - right   * insetWU,
            center + forward * insetWU,
            center - forward * insetWU,
        };

        int clearCount = 0;
        bool centerClear = false;

        for (int i = 0; i < targets.Length; i++)
        {
            bool ok = RayClearHeightAware(a, targets[i], losMask, ignoreRoot);
            if (ok) { clearCount++; if (i == 0) centerClear = true; }
        }

        switch (acceptance)
```

```
        {
            case LoSAcceptance.CenterOnly:   return centerClear;
            case LoSAcceptance.CenterAndAny: return centerClear && clearCount >= 2;
            case LoSAcceptance.Majority:     return clearCount >= 3;     // 3/5
            case LoSAcceptance.All:          return clearCount == targets.Length;
            default:                         return clearCount > 0;      // Any
        }
    }

    private static bool RayClearHeightAware(Vector3 a, Vector3 b, LayerMask mask, Transform ignoreRoot)
    {
        Vector3 d = b - a;
        float L = d.magnitude;
        if (L <= EPS) return true;

        // Kerää kaikki osumat ja käy läpi lähimmästä kaukaisimpaan
        var hits = Physics.RaycastAll(a, d / L, L - EPS, mask, QueryTriggerInteraction.Ignore);
        if (hits == null || hits.Length == 0) return true;

        Array.Sort(hits, (x, y) => x.distance.CompareTo(y.distance));

        float eyeY = a.y;

        foreach (var h in hits)
        {
            if (ignoreRoot != null && h.transform != null && h.transform.IsChildOf(ignoreRoot))
                continue;

            // Jos objektissa on oma “älykäs” looginen tarkistin, käytä sitä
            var dyn = h.collider.GetComponentInParent<IDynamicLoSBlocker>();
            if (dyn != null)
            {
                if (dyn.IsBlockingAtHeight(eyeY))
                    return false; // blokkaa
                else
                    continue;     // ei blokkaa → jatka seuraavaan osumaan
            }

            // Muuten: käytä colliderin ylärajaa
            var topY = h.collider.bounds.max.y;
            if (topY >= eyeY - 0.01f)
                return false; // tarpeeksi korkea → blokkaa
            // Muuten matala → ei blokkaa, jatka
        }

        // Yksikään osuma ei ollut “tarpeeksi korkea”
        return true;
    }

    public static HashSet<GridPosition> ComputeVisibleTilesRaycastHeightAware(
        GridPosition origin,
        int maxRange,
```

```
        LayerMask losMask,
        float eyeHeight = 1.6f,
        int samplesPerCell = 1,
        float insetWU = 0.30f,
        Transform ignoreRoot = null)
    {
        var set = new HashSet<GridPosition>();
        var lg  = LevelGrid.Instance;
        if (lg == null) return set;

        for (int dx = -maxRange; dx <= maxRange; dx++)
        for (int dz = -maxRange; dz <= maxRange; dz++)
        {
            if (dx == 0 && dz == 0) continue; // ei lisätä omaa ruutua
            var gp = origin + new GridPosition(dx, dz, 0);
            if (!lg.IsValidGridPosition(gp)) continue;

            // Sama rengas-metriikka kuin ampumarangeissa (10 * range)
            int cost = SircleCalculator.Sircle(dx, dz);
            if (cost > 10 * maxRange) continue;

            if (HasLineOfSightRaycastHeightAware(origin, gp, losMask, eyeHeight, samplesPerCell, insetWU, ignoreRoot))
                set.Add(gp);
        }
        return set;
    }

}
```

Assets/scripts/Visibility/VisibilitySystem.cs

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


public class VisibilitySystem : MonoBehaviour
{
    [SerializeField] private float pollInterval = 0.25f;
    [SerializeField] private bool hideEnemiesOnStart = true;

    private readonly HashSet<Unit> _visibleNow = new();
    private int _myTeam = -1;
    private bool _didInitialBaseline;

    void OnEnable()
    {
        var tvs = TeamVisionService.Instance;
        if (tvs != null) tvs.OnTeamVisionChanged += HandleTeamVisionChanged;
        StartCoroutine(Co_Poll());
    }

    void OnDisable()
    {
        var tvs = TeamVisionService.Instance;
        if (tvs != null) tvs.OnTeamVisionChanged -= HandleTeamVisionChanged;
        StopAllCoroutines();
        _visibleNow.Clear();
        _didInitialBaseline = false;
        _myTeam = -1;
    }

    private int ResolveLocalTeam()
    {
        // 1) Normaalisti: käytä keskitettyä apuria
        int id = NetworkSync.GetLocalPlayerTeamId(GameModeManager.SelectedMode);

        // 2) Jos ollaan puhdas client (ei host) JA tiimi on vielä epäselvä,
        //    yritä päätellä se ensimmäisestä omistetusta unitista.
        if (NetworkSync.IsClientOnly && UnitManager.Instance != null)
        {
            foreach (var u in UnitManager.Instance.GetAllUnitList())
            {
                if (!u) continue;
                var ni = u.GetComponent<Mirror.NetworkIdentity>();
                if (NetworkSync.IsOwnedHere(ni))
                    return u.GetTeamID(); // Versus: clientille 1, Co-op: 0
            }
        }

        return id;
```

```
    }

    private IEnumerator Co_Poll()
    {
        yield return null;

        _myTeam = ResolveLocalTeam();

        if (hideEnemiesOnStart && !_didInitialBaseline)
        {
            HideAllEnemiesImmediate();
            _didInitialBaseline = true;
        }

        RefreshVisibleEnemies();

        var wait = new WaitForSeconds(pollInterval);
        while (true)
        {
            RefreshVisibleEnemies();
            yield return wait;
        }
    }

    private void HandleTeamVisionChanged(int teamId)
    {
        if (teamId == _myTeam) RefreshVisibleEnemies();
    }

    private void RefreshVisibleEnemies()
    {
        // 0) Hae AJANTASAINEN tiimi jokaisella kutsulla
        int myTeam = NetworkSync.GetLocalPlayerTeamId(GameModeManager.SelectedMode);

        // 0b) Fallback: client-only + Versus → päättele tiimi omistetusta unitista,
        //     jos myTeam vielä 0 (hostin tiimi) ja omia unitteja on jo olemassa.
        if (NetworkSync.IsClientOnly && GameModeManager.SelectedMode == GameMode.Versus && myTeam == 0)
        {
            var um = UnitManager.Instance;
            if (um != null)
            {
                foreach (var unit in um.GetAllUnitList())
                {
                    if (!unit) continue;
                    var ni = unit.GetComponent<Mirror.NetworkIdentity>();
                    if (NetworkSync.IsOwnedHere(ni)) { myTeam = unit.GetTeamID(); break; }
                }
            }
        }

        // (valinnainen) pidä kenttä vain logeja varten ajantasalla
        _myTeam = myTeam;
```

```
        if (myTeam < 0) return;

        var tvs = TeamVisionService.Instance;
        if (tvs == null) { Debug.LogWarning("[VisibilitySystem] TeamVisionService is NULL!"); return; }

        var unitManager = UnitManager.Instance;
        if (unitManager == null) { Debug.LogWarning("[VisibilitySystem] UnitManager is NULL!"); return; }

        var units = unitManager.GetAllUnitList();

        int ownCount = 0, enemyCount = 0, visibleEnemyCount = 0;

        foreach (var unit in units)
        {
            if (!unit) continue;

            int unitTeam = GetTeamId(unit);

            var ni = unit.GetComponent<Mirror.NetworkIdentity>();
            bool isOwn = NetworkSync.IsOwnedHere(ni) || (unitTeam == myTeam); // ← OMISTUS TURVAVERKKO

            if (isOwn)
            {
                ownCount++;
                if (!_visibleNow.Contains(unit))
                {
                    _visibleNow.Add(unit);
                    SetLocallyVisible(unit, true);
                }
                continue;
            }

            enemyCount++;
            var gp = unit.GetGridPosition();
            bool isVisible = tvs.IsVisibleToTeam(myTeam, gp); // ← käytä myTeam
            bool already = _visibleNow.Contains(unit);

            if (isVisible && !already)
            {
                _visibleNow.Add(unit);
                SetLocallyVisible(unit, true);
                visibleEnemyCount++;
            }
            else if (!isVisible && already)
            {
                _visibleNow.Remove(unit);
                SetLocallyVisible(unit, false);
            }
            else if (!isVisible && !already)
            {
                SetLocallyVisible(unit, false);
```

```
            }
        }

    }

    private void HideAllEnemiesImmediate()
    {
        var units = Object.FindObjectsByType<Unit>(FindObjectsSortMode.None);
        int hiddenCount = 0;

        foreach (var u in units)
        {
            if (!u) continue;

            int unitTeam = GetTeamId(u);

            if (unitTeam == _myTeam)
            {
                continue;
            }

            SetLocallyVisible(u, false);
            _visibleNow.Remove(u);
            hiddenCount++;
        }
    }

    private static void SetLocallyVisible(Unit u, bool visible)
    {
        if (!u) return;
        var lv = u.GetComponent<LocalVisibility>();
        if (!lv) lv = u.gameObject.AddComponent<LocalVisibility>();
        lv.Apply(visible);
    }

    private static int GetTeamId(Unit u) => u.GetTeamID();
}
```