1. PrayerTimeSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class PrayerTimeSystem : MonoBehaviour
{
    [BurstCompile]
    struct PrayerTimeCalculation : IJob
    {
        public float latitude;
        public float longitude;
        public int dayOfYear;

        public void Execute()
        {
            // Maldivian prayer time calculations
            CalculateFajr();
            CalculateSunrise();
            CalculateDhuhr();
            CalculateAsr();
            CalculateMaghrib();
            CalculateIsha();
        }

        [BurstCompile]
        void CalculateFajr()
        {
            // Fajr = Dhuhr - T(φ)
            // Where T(φ) = 1/15 * arccos(-tan(φ) * tan(δ))
        }

        [BurstCompile]
        void CalculateDhuhr()
        {
            // Dhuhr = 12 + TimeZone - longitude/15 - equationOfTime
        }

        [BurstCompile]
        void CalculateMaghrib()
        {
            // Maghrib = Dhuhr + T(φ)
        }
```

```
    [BurstCompile]
    void CalculateIsha()
    {
        // Isha = Maghrib + 1.5 hours (Maldivian tradition)
    }

    [BurstCompile]
    void CalculateAsr()
    {
        // Asr = Dhuhr + T(φ) * 1.25 (Shafi'i school)
    }

    [BurstCompile]
    void CalculateSunrise()
    {
        // Sunrise = Dhuhr - T(φ)
    }
}

public static PrayerTimeSystem Instance { get; private set; }

void Awake()
{
    Instance = this;
    SetupMaldivianPrayerTimes();
}

void SetupMaldivianPrayerTimes()
{
    // Maldives coordinates: 3.2028° N, 73.2207° E
    float maldivesLat = 3.2028f;
    float maldivesLng = 73.2207f;

    var job = new PrayerTimeCalculation
    {
        latitude = maldivesLat,
        longitude = maldivesLng,
        dayOfYear = System.DateTime.Now.DayOfYear
    };

    JobHandle handle = job.Schedule();
    handle.Complete();
}

public string GetDhivehiPrayerName(PrayerType prayer)
{
    return prayer switch
    {
```

```
            PrayerType.Fajr => "ﻓَﺠْﺮ",
            PrayerType.Sunrise => "ﺷُﺮُﻭﻕ",
            PrayerType.Dhuhr => "ﻇُﻬْﺮ",
            PrayerType.Asr => "ﻋَﺼْﺮ",
            PrayerType.Maghrib => "ﻣَﻐْﺮِﺏ",
            PrayerType.Isha => "ﻋِﺸَﺎ",
            _ => ""
        };
    }

    public enum PrayerType { Fajr, Sunrise, Dhuhr, Asr, Maghrib, Isha }
}
```

2. WeatherSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class WeatherSystem : MonoBehaviour
{
    [BurstCompile]
    struct MonsoonSimulation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> positions;
        [WriteOnly] public NativeArray<float> precipitation;
        [WriteOnly] public NativeArray<float3> windDirection;

        public float time;
        public float monsoonIntensity;

        public void Execute(int index)
        {
            float3 pos = positions[index];

            // Northeast monsoon (November-April)
            if (IsNortheastMonsoon(time))
            {
                windDirection[index] = new float3(-0.8f, 0, 0.6f) * mon
soonIntensity;
                precipitation[index] = CalculatePrecipitation(pos, Mons
oonType.Northeast);
            }
```

```
            // Southwest monsoon (May-October)
            else
            {
                windDirection[index] = new float3(0.8f, 0, -0.6f) * mon
soonIntensity;
                precipitation[index] = CalculatePrecipitation(pos, Mons
oonType.Southwest);
            }
        }

        [BurstCompile]
        bool IsNortheastMonsoon(float t)
        {
            float month = (t % 365f) / 30.4f;
            return month < 4 || month > 10;
        }

        [BurstCompile]
        float CalculatePrecipitation(float3 pos, MonsoonType type)
        {
            float baseRain = type == MonsoonType.Southwest ? 0.8f : 0.3
f;
            float altitudeEffect = math.max(0, pos.y * 0.001f);
            return baseRain + altitudeEffect + math.sin(pos.x * 0.01f +
 time) * 0.2f;
        }

        enum MonsoonType { Northeast, Southwest }
    }

    public static WeatherSystem Instance { get; private set; }

    void Awake()
    {
        Instance = this;
        InitializeWeatherSimulation();
    }

    void InitializeWeatherSimulation()
    {
        // Initialize with 1000 weather points across Maldives
        int weatherPoints = 1000;
        var positions = new NativeArray<float3>(weatherPoints, Allocato
r.Persistent);
        var precipitation = new NativeArray<float>(weatherPoints, Alloc
ator.Persistent);
        var windDirections = new NativeArray<float3>(weatherPoints, All
ocator.Persistent);
```

```
        // Generate weather grid
        for (int i = 0; i < weatherPoints; i++)
        {
            float lat = 7f + (i % 32) * 0.1f;
            float lng = 72f + (i / 32) * 0.1f;
            positions[i] = new float3(lat, 0, lng);
        }

        var job = new MonsoonSimulation
        {
            positions = positions,
            precipitation = precipitation,
            windDirection = windDirections,
            time = Time.time,
            monsoonIntensity = 1.0f
        };

        JobHandle handle = job.Schedule(weatherPoints, 64);
        handle.Complete();

        positions.Dispose();
        precipitation.Dispose();
        windDirections.Dispose();
    }
}
```

3. BoduberuSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using Unity.Audio;

[BurstCompile]
public class BoduberuSystem : MonoBehaviour
{
    [BurstCompile]
    struct BoduberuRhythmGenerator : IJob
    {
        public NativeArray<float> rhythmPattern;
        public float tempo;
        public int beats;

        public void Execute()
        {
            // Traditional Boduberu rhythm patterns
```

```
            GenerateRhythmPattern();
        }

        [BurstCompile]
        void GenerateRhythmPattern()
        {
            // 8-beat Boduberu pattern: X . X X . X . X
            for (int i = 0; i < beats; i++)
            {
                rhythmPattern[i] = (i % 8) switch
                {
                    0 or 2 or 3 or 5 or 7 => 1.0f, // Strong beats
                    _ => 0.3f // Weak beats
                };
            }
        }
    }

    [BurstCompile]
    struct BoduberuSyncJob : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> rhythmPattern;
        [WriteOnly] public NativeArray<float> syncLevels;
        public float currentTime;
        public float tempo;

        public void Execute(int index)
        {
            float beatTime = currentTime * tempo / 60f;
            int currentBeat = (int)(beatTime * rhythmPattern.Length) %
rhythmPattern.Length;
            syncLevels[index] = rhythmPattern[currentBeat] * math.sin(b
eatTime * math.PI * 2);
        }
    }

    public static BoduberuSystem Instance { get; private set; }
    public AudioSource boduberuAudio;

    void Awake()
    {
        Instance = this;
        InitializeBoduberuSystem();
    }

    void InitializeBoduberuSystem()
    {
        int participants = 12; // Traditional Boduberu group size
        var rhythmPattern = new NativeArray<float>(8, Allocator.TempJob
```

```
);
        var syncLevels = new NativeArray<float>(participants, Allocator
.TempJob);

        var rhythmJob = new BoduberuRhythmGenerator
        {
            rhythmPattern = rhythmPattern,
            tempo = 120f,
            beats = 8
        };

        var syncJob = new BoduberuSyncJob
        {
            rhythmPattern = rhythmPattern,
            syncLevels = syncLevels,
            currentTime = Time.time,
            tempo = 120f
        };

        JobHandle rhythmHandle = rhythmJob.Schedule();
        JobHandle syncHandle = syncJob.Schedule(participants, 1, rhythm
Handle);
        syncHandle.Complete();

        rhythmPattern.Dispose();
        syncLevels.Dispose();

        SetupBoduberuAudio();
    }

    void SetupBoduberuAudio()
    {
        boduberuAudio = gameObject.AddComponent<AudioSource>();
        boduberuAudio.clip = GenerateBoduberuClip();
        boduberuAudio.loop = true;
    }

    AudioClip GenerateBoduberuClip()
    {
        int sampleRate = 44100;
        int length = sampleRate * 8; // 8-second loop
        float[] samples = new float[length];

        // Generate traditional Boduberu drum sounds
        for (int i = 0; i < length; i++)
        {
            float t = i / (float)sampleRate;
            float beat = t * 1.5f; // 90 BPM
```

```
            // Low frequency drum (main beat)
            float lowDrum = Mathf.Sin(2 * Mathf.PI * 80 * t) * Mathf.Ex
p(-t % 1 * 3);

            // High frequency drum (syncopation)
            float highDrum = Mathf.Sin(2 * Mathf.PI * 200 * t) * Mathf.
Exp(-t % 0.5f * 5);

            samples[i] = (lowDrum + highDrum * 0.5f) * 0.3f;
        }

        AudioClip clip = AudioClip.Create("BoduberuLoop", length, 1, sa
mpleRate, false);
        clip.SetData(samples, 0);
        return clip;
    }
}
```

4. IslamicCalendar.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class IslamicCalendar : MonoBehaviour
{
    [BurstCompile]
    struct HijriDateCalculation : IJob
    {
        public int gregorianDay;
        public int gregorianMonth;
        public int gregorianYear;

        [WriteOnly] public NativeArray<int> hijriDate;

        public void Execute()
        {
            // Convert Gregorian to Hijri
            ConvertToHijri();
        }

        [BurstCompile]
        void ConvertToHijri()
        {
            // Simplified Hijri conversion
```

```
            int gYear = gregorianYear;
            int gMonth = gregorianMonth;
            int gDay = gregorianDay;

            // Approximate conversion (simplified for game)
            int hijriYear = gYear - 622 + (gMonth > 7 ? 1 : 0);
            int hijriMonth = gMonth + 3;
            if (hijriMonth > 12) hijriMonth -= 12;

            int hijriDay = gDay;

            hijriDate[0] = hijriDay;
            hijriDate[1] = hijriMonth;
            hijriDate[2] = hijriYear;
        }
    }

    [BurstCompile]
    struct IslamicEventDetection : IJobParallelFor
    {
        [ReadOnly] public NativeArray<int> hijriDate;
        [WriteOnly] public NativeArray<bool> isIslamicEvent;

        public void Execute(int index)
        {
            int day = hijriDate[0];
            int month = hijriDate[1];

            // Check for important Islamic events
            isIslamicEvent[index] = IsRamadan(month) || IsEid(day, mont
h) || IsMawlid(day, month);
        }

        [BurstCompile]
        bool IsRamadan(int month) => month == 9;

        [BurstCompile]
        bool IsEid(int day, int month)
        {
            return (day == 1 && month == 10) || // Eid al-Fitr
                   (day == 10 && month == 12);  // Eid al-Adha
        }

        [BurstCompile]
        bool IsMawlid(int day, int month) => day == 12 && month == 3;
    }

    public static IslamicCalendar Instance { get; private set; }
```

```
    void Awake()
    {
        Instance = this;
        InitializeIslamicCalendar();
    }

    void InitializeIslamicCalendar()
    {
        var today = System.DateTime.Now;
        var hijriDate = new NativeArray<int>(3, Allocator.TempJob);
        var isEvent = new NativeArray<bool>(1, Allocator.TempJob);

        var conversionJob = new HijriDateCalculation
        {
            gregorianDay = today.Day,
            gregorianMonth = today.Month,
            gregorianYear = today.Year,
            hijriDate = hijriDate
        };

        var eventJob = new IslamicEventDetection
        {
            hijriDate = hijriDate,
            isIslamicEvent = isEvent
        };

        JobHandle conversionHandle = conversionJob.Schedule();
        JobHandle eventHandle = eventJob.Schedule(1, 1, conversionHandl
e);
        eventHandle.Complete();

        Debug.Log($"Today in Hijri: {hijriDate[0]}/{hijriDate[1]}/{hijr
iDate[2]}");
        Debug.Log($"Islamic Event Today: {isEvent[0]}");

        hijriDate.Dispose();
        isEvent.Dispose();
    }

    public string GetIslamicMonthName(int month)
    {
        return month switch
        {
            1 => "Muḥarram",
            2 => "Ṣafar",
            3 => "Rabīʿ al-awwal",
            4 => "Rabīʿ ath-thānī",
            5 => "Jumādá al-ūlá",
```

```
            6 => "Jumādá al-ākhirah",
            7 => "Rajab",
            8 => "Shaʿbān",
            9 => "Ramaḍān",
            10 => "Shawwāl",
            11 => "Dhū al-Qaʿdah",
            12 => "Dhū al-Ḥijjah",
            _ => ""
        };
    }
}
```

5. FishingSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class FishingSystem : MonoBehaviour
{
    [BurstCompile]
    struct TraditionalFishingCalculation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> fishingSpots;
        [ReadOnly] public NativeArray<float> tideLevels;
        [WriteOnly] public NativeArray<float> fishProbability;
        [WriteOnly] public NativeArray<int> fishTypes;

        public float timeOfDay;
        public float weatherCondition;

        public void Execute(int index)
        {
            float3 spot = fishingSpots[index];
            float tide = tideLevels[index];

            // Traditional Maldivian fishing knowledge
            fishProbability[index] = CalculateFishProbability(spot, tid
e);
            fishTypes[index] = DetermineFishType(spot, timeOfDay);
        }

        [BurstCompile]
        float CalculateFishProbability(float3 spot, float tide)
        {
```

```
            float baseProbability = 0.3f;

            // Tide affects fishing
            float tideBonus = math.abs(tide) < 0.5f ? 0.4f : 0.1f;

            // Time of day affects different fish
            float timeBonus = (timeOfDay > 6 && timeOfDay < 10) ||
                              (timeOfDay > 16 && timeOfDay < 19) ? 0.3f
: 0.1f;

            // Weather condition
            float weatherBonus = weatherCondition > 0.7f ? -0.2f : 0.2f
;

            return math.clamp(baseProbability + tideBonus + timeBonus +
 weatherBonus, 0f, 1f);
        }

        [BurstCompile]
        int DetermineFishType(float3 spot, float time)
        {
            // Traditional Maldivian fish types
            bool isMorning = time > 5 && time < 9;
            bool isEvening = time > 17 && time < 21;

            if (isMorning)
                return 1; // Skipjack tuna (ކަނޑުއަޑު)
            else if (isEvening)
                return 2; // Yellowfin tuna (ކަނޑުހިއްކަ)
            else
                return 3; // Reef fish (ގަނޑުމަސް)
        }
    }

    [BurstCompile]
    struct PoleAndLineFishing : IJob
    {
        public NativeArray<float> fishCaught;
        public float skillLevel;
        public float baitQuality;
        public float patience;

        public void Execute()
        {
            // Traditional pole and line fishing
            float successRate = skillLevel * 0.4f + baitQuality * 0.3f
+ patience * 0.3f;
            fishCaught[0] = Unity.Mathematics.math.floor(successRate *
```

```
10);
        }
    }

    public static FishingSystem Instance { get; private set; }

    void Awake()
    {
        Instance = this;
        InitializeFishingSystem();
    }

    void InitializeFishingSystem()
    {
        int fishingLocations = 50; // Around 41 islands
        var spots = new NativeArray<float3>(fishingLocations, Allocator
.TempJob);
        var tides = new NativeArray<float>(fishingLocations, Allocator.
TempJob);
        var probabilities = new NativeArray<float>(fishingLocations, Al
locator.TempJob);
        var fishTypes = new NativeArray<int>(fishingLocations, Allocato
r.TempJob);

        // Generate fishing spots around islands
        for (int i = 0; i < fishingLocations; i++)
        {
            float lat = 3.0f + (i % 7) * 0.5f;
            float lng = 73.0f + (i / 7) * 0.5f;
            spots[i] = new float3(lat, 0, lng);
            tides[i] = math.sin(Time.time * 0.1f + i) * 2f; // Tidal si
mulation
        }

        var fishingJob = new TraditionalFishingCalculation
        {
            fishingSpots = spots,
            tideLevels = tides,
            fishProbability = probabilities,
            fishTypes = fishTypes,
            timeOfDay = Time.time % 24,
            weatherCondition = WeatherSystem.Instance?.GetCurrentWeathe
r() ?? 0.5f
        };

        var poleFishingJob = new PoleAndLineFishing
        {
            fishCaught = new NativeArray<float>(1, Allocator.TempJob),
            skillLevel = 0.7f,
```

```
            baitQuality = 0.8f,
            patience = 0.6f
        };

        JobHandle fishingHandle = fishingJob.Schedule(fishingLocations,
 8);
        JobHandle poleHandle = poleFishingJob.Schedule(fishingHandle);
        poleHandle.Complete();

        spots.Dispose();
        tides.Dispose();
        probabilities.Dispose();
        fishTypes.Dispose();
    }

    public string GetDhivehiFishName(int fishType)
    {
        return fishType switch
        {
            1 => "ކަނޑުވަޅު", // Skipjack tuna
            2 => "ކަނޑިދުރަ", // Yellowfin tuna
            3 => "ފަހަރު", // Reef fish
            4 => "ދިވެހި", // Maldivian fish
            _ => "ވަޅު" // Generic fish
        };
    }
}
```

---

6. OceanSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class OceanSystem : MonoBehaviour
{
    [BurstCompile]
    struct TidalSimulation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> oceanPoints;
        [WriteOnly] public NativeArray<float> tideLevels;
        [WriteOnly] public NativeArray<float3> currentFlows;
```

```
        public float time;
        public float lunarPhase;

        public void Execute(int index)
        {
            float3 point = oceanPoints[index];

            // Maldivian tidal patterns (semidiurnal)
            float tide = CalculateTide(point, time);
            float3 current = CalculateCurrent(point, tide);

            tideLevels[index] = tide;
            currentFlows[index] = current;
        }

        [BurstCompile]
        float CalculateTide(float3 point, float t)
        {
            // Semidiurnal tide: 2 high, 2 low per day
            float M2 = 2.0f * math.sin(2 * math.PI * t / 12.42f); // Pr
incipal lunar
            float S2 = 0.3f * math.sin(2 * math.PI * t / 12.0f);  // Pr
incipal solar
            float N2 = 0.2f * math.sin(2 * math.PI * t / 12.66f); // La
rger lunar elliptic

            return M2 + S2 + N2; // Combined tidal constituents
        }

        [BurstCompile]
        float3 CalculateCurrent(float3 point, float tide)
        {
            // Current flows with tide changes
            float speed = math.abs(tide) * 0.5f;
            float angle = math.atan2(point.z, point.x) + tide * 0.1f;

            return new float3(math.cos(angle) * speed, 0, math.sin(angl
e) * speed);
        }
    }

    [BurstCompile]
    struct WaveSimulation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> surfacePoints;
        [WriteOnly] public NativeArray<float> waveHeight;
```

```
        public float time;
        public float windSpeed;

        public void Execute(int index)
        {
            float3 point = surfacePoints[index];

            // Wind wave generation
            float wave1 = math.sin(point.x * 0.1f + time * 2) * 0.5f;
            float wave2 = math.sin(point.z * 0.15f + time * 1.5f) * 0.3
f;
            float wave3 = math.sin((point.x + point.z) * 0.08f + time *
 2.5f) * 0.2f;

            waveHeight[index] = (wave1 + wave2 + wave3) * windSpeed;
        }
    }

    public static OceanSystem Instance { get; private set; }

    void Awake()
    {
        Instance = this;
        InitializeOceanSimulation();
    }

    void InitializeOceanSimulation()
    {
        int oceanGrid = 1000; // 1000 ocean monitoring points
        var points = new NativeArray<float3>(oceanGrid, Allocator.TempJ
ob);
        var tides = new NativeArray<float>(oceanGrid, Allocator.TempJob
);
        var currents = new NativeArray<float3>(oceanGrid, Allocator.Tem
pJob);
        var waves = new NativeArray<float>(oceanGrid, Allocator.TempJob
);

        // Generate ocean grid around Maldives
        for (int i = 0; i < oceanGrid; i++)
        {
            float lat = 2f + (i % 50) * 0.1f;
            float lng = 72f + (i / 50) * 0.2f;
            points[i] = new float3(lat, 0, lng);
        }

        var tideJob = new TidalSimulation
        {
            oceanPoints = points,
```

```
            tideLevels = tides,
            currentFlows = currents,
            time = Time.time / 3600f, // Convert to hours
            lunarPhase = CalculateLunarPhase()
        };

        var waveJob = new WaveSimulation
        {
            surfacePoints = points,
            waveHeight = waves,
            time = Time.time,
            windSpeed = WeatherSystem.Instance?.GetWindSpeed() ?? 1.0f
        };

        JobHandle tideHandle = tideJob.Schedule(oceanGrid, 64);
        JobHandle waveHandle = waveJob.Schedule(oceanGrid, 64, tideHand
le);
        waveHandle.Complete();

        points.Dispose();
        tides.Dispose();
        currents.Dispose();
        waves.Dispose();
    }

    float CalculateLunarPhase()
    {
        // Simplified lunar phase calculation
        float lunarCycle = 29.53f; // days
        float daysSinceNew = (Time.time / 86400f) % lunarCycle;
        return daysSinceNew / lunarCycle;
    }

    public float GetTideLevel(float latitude, float longitude)
    {
        // Sample tide at specific location
        return math.sin(Time.time / 22350f + latitude * 10 + longitude
* 5) * 2.0f;
    }
}
```

7. IslandGenerator.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
```

```
using Unity.Mathematics.Noise;

[BurstCompile]
public class IslandGenerator : MonoBehaviour
{
    [BurstCompile]
    struct IslandShapeGeneration : IJobParallelFor
    {
        [WriteOnly] public NativeArray<float> islandHeight;
        [ReadOnly] public NativeArray<float2> positions;

        public int islandIndex;
        public float seed;

        public void Execute(int index)
        {
            float2 pos = positions[index];

            // Generate realistic island shapes using Perlin noise
            float height = GenerateIslandShape(pos, islandIndex, seed);
            islandHeight[index] = height;
        }

        [BurstCompile]
        float GenerateIslandShape(float2 pos, int island, float seed)
        {
            // Base island shape
            float dist = math.length(pos);
            float islandMask = math.saturate(1.0f - dist * 0.5f);

            // Add fractal noise for realistic terrain
            float noise = 0;
            float frequency = 1.0f;
            float amplitude = 1.0f;

            for (int i = 0; i < 4; i++)
            {
                noise += snoise(pos * frequency + seed) * amplitude;
                frequency *= 2.0f;
                amplitude *= 0.5f;
            }

            return islandMask * noise * 10.0f;
        }
    }

    [BurstCompile]
    struct AtollGeneration : IJob
    {
```

```
        [WriteOnly] public NativeArray<float3> atollPositions;
        public int atollCount;
        public float seed;

        public void Execute()
        {
            // Generate real Maldivian atoll positions
            GenerateMaldivianAtolls();
        }

        [BurstCompile]
        void GenerateMaldivianAtolls()
        {
            // Real atoll coordinates (simplified)
            float3[] realAtolls = new float3[]
            {
                new float3(7.15f, 0, 72.9f),   // Haa Alif
                new float3(6.8f, 0, 73.1f),    // Haa Dhaal
                new float3(6.4f, 0, 73.2f),    // Shaviyani
                new float3(6.1f, 0, 73.3f),    // Noonu
                new float3(5.8f, 0, 73.4f),    // Raa
                new float3(5.5f, 0, 73.0f),    // Baa
                new float3(5.2f, 0, 73.1f),    // Lhaviyani
                new float3(4.9f, 0, 73.3f),    // Kaafu
                new float3(4.6f, 0, 73.4f),    // Alif Alif
                new float3(4.3f, 0, 73.5f),    // Alif Dhaal
                new float3(4.0f, 0, 73.3f),    // Vaavu
                new float3(3.7f, 0, 73.4f),    // Meemu
                new float3(3.4f, 0, 73.2f),    // Faafu
                new float3(3.1f, 0, 73.3f),    // Dhaalu
                new float3(2.8f, 0, 73.1f),    // Thaa
                new float3(2.5f, 0, 73.0f),    // Laamu
                new float3(2.2f, 0, 73.2f),    // Gaafu Alif
                new float3(1.9f, 0, 73.3f),    // Gaafu Dhaal
                new float3(1.6f, 0, 73.4f),    // Gnaviyani
                new float3(1.3f, 0, 73.5f)     // Addu
            };

            for (int i = 0; i < math.min(atollCount, realAtolls.Length)
; i++)
            {
                atollPositions[i] = realAtolls[i];
            }
        }
    }

    public static IslandGenerator Instance { get; private set; }
    public GameObject[] islandPrefabs;
```

```
    void Awake()
    {
        Instance = this;
        GenerateAllIslands();
    }

    void GenerateAllIslands()
    {
        const int totalIslands = 41; // Real Maldives islands
        const int atolls = 20; // Real Maldivian atolls

        var atollPositions = new NativeArray<float3>(atolls, Allocator.
TempJob);
        var islandHeights = new NativeArray<float>(1000, Allocator.Temp
Job);
        var positions = new NativeArray<float2>(1000, Allocator.TempJob
);


        // Generate atoll positions
        var atollJob = new AtollGeneration
        {
            atollPositions = atollPositions,
            atollCount = atolls,
            seed = Time.time
        };

        // Generate height map for each island
        for (int island = 0; island < totalIslands; island++)
        {
            for (int i = 0; i < 1000; i++)
            {
                float x = (i % 50) * 0.1f - 2.5f;
                float y = (i / 50) * 0.1f - 2.5f;
                positions[i] = new float2(x, y);
            }

            var islandJob = new IslandShapeGeneration
            {
                islandHeight = islandHeights,
                positions = positions,
                islandIndex = island,
                seed = island * 1000 + Time.time
            };

            JobHandle handle = islandJob.Schedule(1000, 64);
            handle.Complete();

            CreateIslandMesh(island, islandHeights, atollPositions[isla
nd % atolls]);
```

```
        }

        atollPositions.Dispose();
        islandHeights.Dispose();
        positions.Dispose();
    }

    void CreateIslandMesh(int islandIndex, NativeArray<float> heights,
float3 atollPosition)
    {
        GameObject island = new GameObject($"Island_{islandIndex}");
        island.transform.position = atollPosition;

        MeshFilter meshFilter = island.AddComponent<MeshFilter>();
        MeshRenderer meshRenderer = island.AddComponent<MeshRenderer>()
;

        // Create mesh from height data
        Mesh islandMesh = GenerateIslandMesh(heights);
        meshFilter.mesh = islandMesh;

        // Add appropriate material
        meshRenderer.material = GetIslandMaterial(islandIndex);

        // Add collision
        MeshCollider collider = island.AddComponent<MeshCollider>();
        collider.sharedMesh = islandMesh;
    }

    Mesh GenerateIslandMesh(NativeArray<float> heights)
    {
        Mesh mesh = new Mesh();
        int size = (int)math.sqrt(heights.Length);

        Vector3[] vertices = new Vector3[heights.Length];
        int[] triangles = new int[(size - 1) * (size - 1) * 6];

        // Create vertices
        for (int i = 0; i < heights.Length; i++)
        {
            int x = i % size;
            int z = i / size;
            vertices[i] = new Vector3(x * 0.1f, heights[i], z * 0.1f);
        }

        // Create triangles
        int triIndex = 0;
        for (int z = 0; z < size - 1; z++)
```

```
        {
            for (int x = 0; x < size - 1; x++)
            {
                int topLeft = z * size + x;
                int topRight = topLeft + 1;
                int bottomLeft = (z + 1) * size + x;
                int bottomRight = bottomLeft + 1;

                triangles[triIndex++] = topLeft;
                triangles[triIndex++] = bottomLeft;
                triangles[triIndex++] = topRight;

                triangles[triIndex++] = topRight;
                triangles[triIndex++] = bottomLeft;
                triangles[triIndex++] = bottomRight;
            }
        }

        mesh.vertices = vertices;
        mesh.triangles = triangles;
        mesh.RecalculateNormals();

        return mesh;
    }

    Material GetIslandMaterial(int islandIndex)
    {
        // Return appropriate material based on island type
        return islandPrefabs[islandIndex % islandPrefabs.Length].GetCom
ponent<MeshRenderer>().sharedMaterial;
    }
}
```

8. FloraSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class FloraSystem : MonoBehaviour
{
    [BurstCompile]
    struct FloraDistribution : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> terrainPoints;
```

```
[ReadOnly] public NativeArray<float> terrainHeight;
[WriteOnly] public NativeArray<int> floraType;
[WriteOnly] public NativeArray<float> floraDensity;

public float salinity;
public float rainfall;
public float temperature;

public void Execute(int index)
{
    float3 point = terrainPoints[index];
    float height = terrainHeight[index];

    // Determine flora based on Maldivian ecosystem
    floraType[index] = DetermineFloraType(point, height);
    floraDensity[index] = CalculateFloraDensity(point, height);
}

[BurstCompile]
int DetermineFloraType(float3 point, float height)
{
    // Maldivian native flora types
    if (height < 0.5f) // Beach area
    {
        if (salinity > 0.8f)
            return 1; // Coconut palm (ގަނޑު)
        else
            return 2; // Sea lettuce (ފަށިގަނޑު)
    }
    else if (height < 2.0f) // Coastal area
    {
        if (rainfall > 0.6f)
            return 3; // Breadfruit tree (ހުންބޭ)
        else
            return 4; // Tropical almond (ހުންބޭ ގަސް)
    }
    else // Inland
    {
        return 5; // Banyan tree (ނިޔޮ ގަސް)
    }
}

[BurstCompile]
float CalculateFloraDensity(float3 point, float height)
{
    float baseDensity = 0.5f;
```

```
            // Salinity affects most plants negatively
            float salinityFactor = math.saturate(1.0f - salinity);

            // Rainfall affects positively
            float rainfallFactor = rainfall;

            // Height affects (coastal vs inland)
            float heightFactor = math.saturate(height / 5.0f);

            return baseDensity * salinityFactor * rainfallFactor * heig
htFactor;
        }
    }

    [BurstCompile]
    struct CoconutPalmGrowth : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> palmPositions;
        [WriteOnly] public NativeArray<float> growthStage;

        public float time;
        public float soilQuality;

        public void Execute(int index)
        {
            float3 pos = palmPositions[index];

            // Coconut palm growth simulation
            growthStage[index] = CalculateGrowthStage(pos, time);
        }

        [BurstCompile]
        float CalculateGrowthStage(float3 pos, float t)
        {
            // Coconut palms take 6-10 years to mature
            float growthTime = 7.0f * 365.25f * 24.0f; // 7 years in ho
urs
            float currentAge = (t + pos.x * 1000) % growthTime;

            return math.saturate(currentAge / growthTime);
        }
    }

    public static FloraSystem Instance { get; private set; }
    public GameObject[] floraPrefabs; // 12 flora types

    // Maldivian native flora
    public enum MaldivianFlora
```

```
    {
        CoconutPalm = 1,      // ދަރަށް
        SeaLettuce = 2,       // ދަށު ލެޓިސް
        BreadfruitTree = 3,   // ހަނާގު
        TropicalAlmond = 4,   // ހަނާގު ކައްޗ
        BanyanTree = 5,       // ނިޔަ ކައްޗ
        Pandanus = 6,         // ކާސްދަށުރ
        SeaHibiscus = 7,      // ދ ހަނާގު
        BeachMorningGlory = 8, // ލަދުގަ
        Ironwood = 9,         // ކާސްދަރަ
        Mangrove = 10,        // ތުނ
        Seagrass = 11,        // ދަށު ކައްޗ
        Coral = 12            // ކުއްރ
    }

    void Awake()
    {
        Instance = this;
        GenerateMaldivianFlora();
    }

    void GenerateMaldivianFlora()
    {
        int terrainPoints = 2000; // High resolution flora distribution
        var points = new NativeArray<float3>(terrainPoints, Allocator.T
empJob);
        var heights = new NativeArray<float>(terrainPoints, Allocator.T
empJob);
        var floraTypes = new NativeArray<int>(terrainPoints, Allocator.
TempJob);
        var densities = new NativeArray<float>(terrainPoints, Allocator
.TempJob);

        // Sample terrain
        for (int i = 0; i < terrainPoints; i++)
        {
            float lat = 2f + (i % 50) * 0.1f;
            float lng = 72f + (i / 50) * 0.2f;
            points[i] = new float3(lat, 0, lng);
            heights[i] = GetTerrainHeight(lat, lng);
        }

        var floraJob = new FloraDistribution
        {
            terrainPoints = points,
```

```
            terrainHeight = heights,
            floraType = floraTypes,
            floraDensity = densities,
            salinity = 0.7f, // High salinity in Maldives
            rainfall = 0.6f, // Moderate rainfall
            temperature = 28.5f // Average temperature
        };

        JobHandle handle = floraJob.Schedule(terrainPoints, 64);
        handle.Complete();

        // Spawn flora based on calculations
        for (int i = 0; i < terrainPoints; i++)
        {
            if (densities[i] > 0.3f) // Threshold for flora spawning
            {
                SpawnFlora(points[i], floraTypes[i], densities[i]);
            }
        }

        points.Dispose();
        heights.Dispose();
        floraTypes.Dispose();
        densities.Dispose();
    }

    void SpawnFlora(float3 position, int floraType, float density)
    {
        if (floraType > 0 && floraType <= floraPrefabs.Length)
        {
            GameObject flora = Instantiate(floraPrefabs[floraType - 1],
 position, Quaternion.identity);
            flora.transform.localScale *= (0.8f + density * 0.4f); // V
ary size based on density

            // Add cultural significance
            AddCulturalSignificance(flora, (MaldivianFlora)floraType);
        }
    }

    void AddCulturalSignificance(GameObject flora, MaldivianFlora type)
    {
        FloraCulture culture = flora.AddComponent<FloraCulture>();
        culture.floraType = type;
        culture.dhivehiName = GetDhivehiFloraName(type);
        culture.traditionalUse = GetTraditionalUse(type);
    }

    string GetDhivehiFloraName(MaldivianFlora flora)
```

```
    {
        return flora switch
        {
            MaldivianFlora.CoconutPalm => "ދަރަށް",
            MaldivianFlora.SeaLettuce => "ހަސް ގަނޑު",
            MaldivianFlora.BreadfruitTree => "ތެރޭ",
            MaldivianFlora.TropicalAlmond => "ތެރޭ ގަސް",
            MaldivianFlora.BanyanTree => "ނިބަ ގަސް",
            MaldivianFlora.Pandanus => "ކާހަނބޫރު",
            MaldivianFlora.SeaHibiscus => "ފޮތެރޭ",
            MaldivianFlora.BeachMorningGlory => "ވިއްޔަ",
            MaldivianFlora.Ironwood => "ކާހަނބަރު",
            MaldivianFlora.Mangrove => "ރަތްލަބު",
            MaldivianFlora.Seagrass => "ހަސް ގަސް",
            MaldivianFlora.Coral => "ކެއްރޮ",
            _ => "ގަސް"
        };
    }

    string GetTraditionalUse(MaldivianFlora flora)
    {
        return flora switch
        {
            MaldivianFlora.CoconutPalm => "Food, oil, construction mate
rial",
            MaldivianFlora.BreadfruitTree => "Staple food, traditional
medicine",
            MaldivianFlora.Pandanus => "Mat weaving, traditional crafts
",
            MaldivianFlora.Ironwood => "Boat building, construction",
            MaldivianFlora.Mangrove => "Coastal protection, fishing hab
itat",
            _ => "Environmental beautification"
        };
    }
}

class FloraCulture : MonoBehaviour
{
    public FloraSystem.MaldivianFlora floraType;
    public string dhivehiName;
    public string traditionalUse;
}
```

9. VehicleSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class VehicleSystem : MonoBehaviour
{
    [BurstCompile]
    struct VehiclePhysics : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> positions;
        [ReadOnly] public NativeArray<float3> velocities;
        [ReadOnly] public NativeArray<float> steeringAngles;
        [WriteOnly] public NativeArray<float3> newPositions;
        [WriteOnly] public NativeArray<float3> newVelocities;

        public float deltaTime;
        public float3 gravity;

        public void Execute(int index)
        {
            float3 pos = positions[index];
            float3 vel = velocities[index];
            float steer = steeringAngles[index];

            // Apply vehicle physics
            float3 acceleration = CalculateAcceleration(vel, steer);
            float3 newVel = vel + acceleration * deltaTime;
            float3 newPos = pos + newVel * deltaTime;

            newVelocities[index] = newVel;
            newPositions[index] = newPos;
        }

        [BurstCompile]
        float3 CalculateAcceleration(float3 velocity, float steering)
        {
            // Simplified vehicle dynamics
            float speed = math.length(velocity);
            float maxSpeed = 15.0f; // m/s (54 km/h)

            // Apply speed limit
            float3 forwardVel = math.normalize(velocity) * math.min(spe
ed, maxSpeed);
```

```
            // Apply steering
            float3 steeringForce = new float3(math.sin(steering), 0, ma
th.cos(steering)) * 2.0f;

            return steeringForce;
        }
    }

    [BurstCompile]
    struct MaldivianTrafficSimulation : IJob
    {
        public NativeArray<float> trafficDensity;
        public float timeOfDay;
        public float dayOfWeek;

        public void Execute()
        {
            // Simulate Maldivian traffic patterns
            trafficDensity[0] = CalculateTrafficDensity();
        }

        [BurstCompile]
        float CalculateTrafficDensity()
        {
            // Maldivian traffic: low in morning, peak in evening
            float morningRush = math.saturate(math.sin((timeOfDay - 7)
* math.PI / 6)) * 0.8f;
            float eveningRush = math.saturate(math.sin((timeOfDay - 17)
 * math.PI / 6)) * 0.9f;

            // Weekend traffic (Friday in Maldives)
            float weekendFactor = dayOfWeek == 5 ? 0.6f : 1.0f;

            return math.max(morningRush, eveningRush) * weekendFactor;
        }
    }

    public static VehicleSystem Instance { get; private set; }

    [System.Serializable]
    public class MaldivianVehicle
    {
        public string name;
        public string dhivehiName;
        public VehicleType type;
        public GameObject prefab;
        public bool isTraditional;
        public float speed;
```

```
        public int passengerCapacity;
    }

    public enum VehicleType
    {
        Car,
        Motorcycle,
        Scooter,
        FishingBoat,
        Speedboat,
        Ferry,
        Seaplane,
        Bicycle,
        Truck,
        Bus
    }

    public MaldivianVehicle[] vehicles; // 40 vehicles

    void Awake()
    {
        Instance = this;
        InitializeVehicleSystem();
    }

    void InitializeVehicleSystem()
    {
        // Initialize 40 Maldivian vehicles
        InitializeVehicleDatabase();

        int activeVehicles = 100; // Active vehicles in scene
        var positions = new NativeArray<float3>(activeVehicles, Allocat
or.TempJob);
        var velocities = new NativeArray<float3>(activeVehicles, Alloca
tor.TempJob);
        var steering = new NativeArray<float>(activeVehicles, Allocator
.TempJob);
        var newPositions = new NativeArray<float3>(activeVehicles, Allo
cator.TempJob);
        var newVelocities = new NativeArray<float3>(activeVehicles, All
ocator.TempJob);

        // Initialize vehicle data
        for (int i = 0; i < activeVehicles; i++)
        {
            positions[i] = new float3(UnityEngine.Random.Range(-10f, 10
f), 0, UnityEngine.Random.Range(-10f, 10f));
            velocities[i] = new float3(UnityEngine.Random.Range(-5f, 5f
), 0, UnityEngine.Random.Range(-5f, 5f));
```

```
            steering[i] = UnityEngine.Random.Range(-1f, 1f);
        }

        var physicsJob = new VehiclePhysics
        {
            positions = positions,
            velocities = velocities,
            steeringAngles = steering,
            newPositions = newPositions,
            newVelocities = newVelocities,
            deltaTime = Time.fixedDeltaTime,
            gravity = new float3(0, -9.81f, 0)
        };

        var trafficJob = new MaldivianTrafficSimulation
        {
            trafficDensity = new NativeArray<float>(1, Allocator.TempJo
b),
            timeOfDay = Time.time % 24,
            dayOfWeek = (int)(Time.time / 24) % 7
        };

        JobHandle physicsHandle = physicsJob.Schedule(activeVehicles, 1
6);
        JobHandle trafficHandle = trafficJob.Schedule(physicsHandle);
        trafficHandle.Complete();

        positions.Dispose();
        velocities.Dispose();
        steering.Dispose();
        newPositions.Dispose();
        newVelocities.Dispose();
    }

    void InitializeVehicleDatabase()
    {
        vehicles = new MaldivianVehicle[40];

        // Traditional Maldivian vehicles
        vehicles[0] = new MaldivianVehicle { name = "Bokkura", dhivehiN
ame = "ދޮށުބޯކަ", type = VehicleType.FishingBoat, isTraditional = true, sp
eed = 15, passengerCapacity = 4 };
        vehicles[1] = new MaldivianVehicle { name = "Dhoni", dhivehiNam
e = "ދޯނި", type = VehicleType.FishingBoat, isTraditional = true, speed
= 20, passengerCapacity = 8 };
        vehicles[2] = new MaldivianVehicle { name = "Sathari Dhoni", dh
ivehiName = "ސަތަރި ދޯނި", type = VehicleType.Ferry, isTraditional = true,
 speed = 25, passengerCapacity = 50 };
```

```
        // Modern vehicles
        vehicles[3] = new MaldivianVehicle { name = "Taxi Car", dhivehi
Name = "ﺗﮑﺴﻰ", type = VehicleType.Car, isTraditional = false, speed = 6
0, passengerCapacity = 4 };
        vehicles[4] = new MaldivianVehicle { name = "Private Car", dhiv
ehiName = "ﭘﺮﺍﻳﻴﭧ ﮐﺎﺭ", type = VehicleType.Car, isTraditional = false, s
peed = 80, passengerCapacity = 5 };
        vehicles[5] = new MaldivianVehicle { name = "Motorcycle", dhive
hiName = "ﻣﻮﭨﺮ ﺳﺎﻳﻜﻞ", type = VehicleType.Motorcycle, isTraditional = fa
lse, speed = 100, passengerCapacity = 2 };

        // Add remaining 34 vehicles...
        for (int i = 6; i < 40; i++)
        {
            vehicles[i] = GenerateRandomVehicle(i);
        }
    }

    MaldivianVehicle GenerateRandomVehicle(int index)
    {
        VehicleType[] types = (VehicleType[])System.Enum.GetValues(type
of(VehicleType));
        VehicleType randomType = types[index % types.Length];

        return new MaldivianVehicle
        {
            name = $"{randomType}_{index}",
            dhivehiName = GetDhivehiVehicleName(randomType),
            type = randomType,
            isTraditional = false,
            speed = UnityEngine.Random.Range(30, 120),
            passengerCapacity = UnityEngine.Random.Range(1, 20)
        };
    }

    string GetDhivehiVehicleName(VehicleType type)
    {
        return type switch
        {
            VehicleType.Car => "ﮐﺎﺭ",
            VehicleType.Motorcycle => "ﻣﻮﭨﺮ ﺳﺎﻳﻜﻞ",
            VehicleType.Scooter => "ﺳﻜﻮﭨﺮ",
            VehicleType.FishingBoat => "ﺩﯗﺱ ﻓﮍﻫﻰ",
            VehicleType.Speedboat => "ﺳﭙﻴﮈ ﺑﯚﭦ",
            VehicleType.Ferry => "ﻓﻪﺭﻯ",
```

```
            VehicleType.Seaplane => "ﺳﻲﻔﻠﻦ",
            VehicleType.Bicycle => "ﻫﺎﺭﺳﻮﺍﺭ",
            VehicleType.Truck => "ﭨﺮﮎ",
            VehicleType.Bus => "ﻫﺴ",
            _ => "ﻭﺍﺭﺍﺭ"
        };
    }
}
```

10. CharacterSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class CharacterSystem : MonoBehaviour
{
    [BurstCompile]
    struct CharacterBehaviorSimulation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> positions;
        [ReadOnly] public NativeArray<float> needs;
        [ReadOnly] public NativeArray<float> schedules;
        [WriteOnly] public NativeArray<float3> destinations;
        [WriteOnly] public NativeArray<float> activities;

        public float timeOfDay;
        public float dayOfWeek;

        public void Execute(int index)
        {
            float3 pos = positions[index];
            float need = needs[index];
            float schedule = schedules[index];

            // Simulate Maldivian daily life
            float3 destination = CalculateDestination(pos, schedule, ti
meOfDay);
            float activity = DetermineActivity(schedule, timeOfDay);

            destinations[index] = destination;
            activities[index] = activity;
        }
```

```
        [BurstCompile]
        float3 CalculateDestination(float3 currentPos, float schedule,
float time)
        {
            // Maldivian daily patterns
            if (time < 5) // Early morning prayer
                return new float3(currentPos.x + 0.1f, 0, currentPos.z
+ 0.1f); // Mosque
            else if (time < 8) // Morning fishing
                return new float3(currentPos.x - 0.5f, 0, currentPos.z)
; // Harbor
            else if (time < 12) // Work/school
                return new float3(currentPos.x + 0.3f, 0, currentPos.z
- 0.2f); // Workplace
            else if (time < 14) // Lunch/rest
                return currentPos; // Home
            else if (time < 17) // Afternoon activities
                return new float3(currentPos.x, 0, currentPos.z + 0.4f)
; // Market
            else if (time < 19) // Evening prayer/family time
                return new float3(currentPos.x + 0.1f, 0, currentPos.z
+ 0.1f); // Mosque then home
            else // Night
                return currentPos; // Home
        }

        [BurstCompile]
        float DetermineActivity(float schedule, float time)
        {
            // Activity types: 1=praying, 2=working, 3=fishing, 4=shopp
ing, 5=socializing
            return time switch
            {
                < 5 => 1, // Praying
                < 8 => 3, // Fishing
                < 12 => 2, // Working
                < 14 => 5, // Socializing/lunch
                < 17 => 4, // Shopping
                < 19 => 1, // Praying
                _ => 5 // Family time
            };
        }
    }

    [BurstCompile]
    struct PopulationDistribution : IJob
    {
        [WriteOnly] public NativeArray<float3> populationCenters;
```

```
        public int population;
        public float islandSize;

        public void Execute()
        {
            GenerateMaldivianPopulationDistribution();
        }

        [BurstCompile]
        void GenerateMaldivianPopulationDistribution()
        {
            // Malé and other major population centers
            float3[] majorCenters = new float3[]
            {
                new float3(4.1755f, 0, 73.5093f), // Malé
                new float3(4.7667f, 0, 73.3f),    // Addu
                new float3(5.2f, 0, 73.0f),       // Hithadhoo
                new float3(4.9f, 0, 73.3f),       // Naifaru
                new float3(5.5f, 0, 73.0f),       // Kulhudhuffushi
                new float3(4.6f, 0, 73.4f),       // Eydhafushi
                new float3(5.8f, 0, 73.4f),       // Ungoofaaru
                new float3(6.1f, 0, 73.3f),       // Funadhoo
                new float3(6.4f, 0, 73.2f),       // Komandoo
                new float3(6.8f, 0, 73.1f)        // Veymandoo
            };

            for (int i = 0; i < math.min(population, majorCenters.Lengt
h); i++)
            {
                populationCenters[i] = majorCenters[i];
            }
        }
    }

    public static CharacterSystem Instance { get; private set; }

    [System.Serializable]
    public class MaldivianCharacter
    {
        public string firstName;
        public string lastName;
        public string dhivehiName;
        public Gender gender;
        public int age;
        public Occupation occupation;
        public string island;
        public bool isPlayer;
        public GameObject prefab;
```

```
        public enum Gender { Male, Female }
        public enum Occupation
        {
            Fisherman, Teacher, GovernmentWorker, Businessman,
            Student, Housewife, TourismWorker, HealthcareWorker,
            ReligiousScholar, Craftsperson, TransportWorker, Unemployed
        }
    }

    public MaldivianCharacter[] characters; // 300+ characters

    void Awake()
    {
        Instance = this;
        GenerateMaldivianPopulation();
    }

    void GenerateMaldivianPopulation()
    {
        const int population = 300; // Representative population
        characters = new MaldivianCharacter[population];

        var positions = new NativeArray<float3>(population, Allocator.T
empJob);
        var needs = new NativeArray<float>(population, Allocator.TempJo
b);
        var schedules = new NativeArray<float>(population, Allocator.Te
mpJob);
        var destinations = new NativeArray<float3>(population, Allocato
r.TempJob);
        var activities = new NativeArray<float>(population, Allocator.T
empJob);

        // Initialize population data
        for (int i = 0; i < population; i++)
        {
            characters[i] = GenerateRandomCharacter(i);
            positions[i] = GetRandomIslandPosition();
            needs[i] = UnityEngine.Random.Range(0f, 1f);
            schedules[i] = UnityEngine.Random.Range(0f, 24f);
        }

        var behaviorJob = new CharacterBehaviorSimulation
        {
            positions = positions,
            needs = needs,
            schedules = schedules,
            destinations = destinations,
            activities = activities,
```

```
            timeOfDay = Time.time % 24,
            dayOfWeek = (int)(Time.time / 24) % 7
        };

        var populationJob = new PopulationDistribution
        {
            populationCenters = new NativeArray<float3>(10, Allocator.T
empJob),
            population = population,
            islandSize = 100f
        };

        JobHandle behaviorHandle = behaviorJob.Schedule(population, 16)
;
        JobHandle populationHandle = populationJob.Schedule(behaviorHan
dle);
        populationHandle.Complete();

        positions.Dispose();
        needs.Dispose();
        schedules.Dispose();
        destinations.Dispose();
        activities.Dispose();
    }

    MaldivianCharacter GenerateRandomCharacter(int index)
    {
        bool isMale = index % 2 == 0;
        string firstName = isMale ? GetMaleName() : GetFemaleName();
        string lastName = GetLastName();

        return new MaldivianCharacter
        {
            firstName = firstName,
            lastName = lastName,
            dhivehiName = GetDhivehiName(firstName, lastName),
            gender = isMale ? MaldivianCharacter.Gender.Male : Maldivia
nCharacter.Gender.Female,
            age = UnityEngine.Random.Range(18, 80),
            occupation = (MaldivianCharacter.Occupation)UnityEngine.Ran
dom.Range(0, 12),
            island = GetRandomIsland(),
            isPlayer = index == 0, // First character is player
            prefab = null // Assign prefab later
        };
    }

    string GetMaleName()
    {
```

```
        string[] names = { "Mohamed", "Ahmed", "Ibrahim", "Ali", "Hassa
n", "Hussein", "Abdulla", "Yoosuf", "Ismail", "Anwar" };
        return names[UnityEngine.Random.Range(0, names.Length)];
    }

    string GetFemaleName()
    {
        string[] names = { "Aminath", "Mariyam", "Fathimath", "Aisha",
"Shifa", "Nashwa", "Layaan", "Azaan", "Rifaath", "Shaima" };
        return names[UnityEngine.Random.Range(0, names.Length)];
    }

    string GetLastName()
    {
        string[] names = { "Abdul Kareem", "Ibrahim", "Hassan", "Ali",
"Mohamed", "Ahmed", "Yoosuf", "Ismail", "Rasheed", "Shafeeq" };
        return names[UnityEngine.Random.Range(0, names.Length)];
    }

    string GetDhivehiName(string firstName, string lastName)
    {
        // Simplified Dhivehi name generation
        return $"ތަޢާރަފް {firstName} {lastName}";
    }

    string GetRandomIsland()
    {
        string[] islands = { "Malé", "Addu", "Hithadhoo", "Naifaru", "K
ulhudhuffushi", "Eydhafushi", "Ungoofaaru", "Funadhoo", "Komandoo", "Ve
ymandoo" };
        return islands[UnityEngine.Random.Range(0, islands.Length)];
    }

    float3 GetRandomIslandPosition()
    {
        float lat = UnityEngine.Random.Range(2f, 7f);
        float lng = UnityEngine.Random.Range(72f, 74f);
        return new float3(lat, 0, lng);
    }
}
```

11. GangSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
```

```
using Unity.Mathematics;

[BurstCompile]
public class GangSystem : MonoBehaviour
{
    [BurstCompile]
    struct GangTerritoryControl : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> territoryCenters;
        [ReadOnly] public NativeArray<float> gangStrengths;
        [WriteOnly] public NativeArray<float> territoryControl;

        public float time;

        public void Execute(int index)
        {
            float3 center = territoryCenters[index];
            float strength = gangStrengths[index];

            // Calculate territory control based on proximity and stren
gth
            territoryControl[index] = CalculateTerritoryInfluence(cente
r, strength);
        }

        [BurstCompile]
        float CalculateTerritoryInfluence(float3 center, float strength
)
        {
            float influence = 0;

            for (int i = 0; i < territoryCenters.Length; i++)
            {
                if (i == index) continue;

                float3 otherCenter = territoryCenters[i];
                float otherStrength = gangStrengths[i];

                float distance = math.length(center - otherCenter);
                float proximityInfluence = math.saturate(1.0f - distanc
e * 0.1f);

                influence += proximityInfluence * (strength - otherStre
ngth);
            }

            return math.saturate(influence * 0.1f + 0.5f);
        }
```

```
    }

    [BurstCompile]
    struct GangActivitySimulation : IJob
    {
        public NativeArray<float> activityLevels;
        public float timeOfDay;
        public float policePresence;

        public void Execute()
        {
            // Simulate gang activities throughout the day
            for (int i = 0; i < activityLevels.Length; i++)
            {
                activityLevels[i] = CalculateActivityLevel(i, timeOfDay
);
            }
        }

        [BurstCompile]
        float CalculateActivityLevel(int gangIndex, float time)
        {
            // Gangs more active at night
            float nightBonus = time < 6 || time > 22 ? 0.5f : 0f;

            // Police presence reduces activity
            float policeReduction = policePresence * 0.3f;

            // Base activity varies by gang
            float baseActivity = 0.3f + gangIndex * 0.01f;

            return math.saturate(baseActivity + nightBonus - policeRedu
ction);
        }
    }

    public static GangSystem Instance { get; private set; }

    [System.Serializable]
    public class MaldivianGang
    {
        public string name;
        public string dhivehiName;
        public string territory;
        public int memberCount;
        public GangType type;
        public float strength;
        public Color gangColor;
        public bool isActive;
```

```
        public string[] activities;

        public enum GangType
        {
            StreetGang,
            DrugCartel,
            SmugglingRing,
            ProtectionRacket,
            CyberGang,
            PoliticalGang,
            PrisonGang,
            YouthGang,
            OrganizedCrime,
            LocalTurf
        }
    }

    public MaldivianGang[] gangs; // 83 gangs

    void Awake()
    {
        Instance = this;
        InitializeGangSystem();
    }

    void InitializeGangSystem()
    {
        const int gangCount = 83; // Total gangs
        gangs = new MaldivianGang[gangCount];

        var territoryCenters = new NativeArray<float3>(gangCount, Alloc
ator.TempJob);
        var gangStrengths = new NativeArray<float>(gangCount, Allocator
.TempJob);
        var territoryControl = new NativeArray<float>(gangCount, Alloca
tor.TempJob);
        var activityLevels = new NativeArray<float>(gangCount, Allocato
r.TempJob);

        // Initialize gangs
        for (int i = 0; i < gangCount; i++)
        {
            gangs[i] = GenerateMaldivianGang(i);
            territoryCenters[i] = GetTerritoryCenter(gangs[i].territory
);
            gangStrengths[i] = gangs[i].strength;
        }

        var territoryJob = new GangTerritoryControl
```

```
        {
            territoryCenters = territoryCenters,
            gangStrengths = gangStrengths,
            territoryControl = territoryControl,
            time = Time.time
        };

        var activityJob = new GangActivitySimulation
        {
            activityLevels = activityLevels,
            timeOfDay = Time.time % 24,
            policePresence = 0.5f
        };

        JobHandle territoryHandle = territoryJob.Schedule(gangCount, 8)
;
        JobHandle activityHandle = activityJob.Schedule(territoryHandle
);
        activityHandle.Complete();

        // Update gang data
        for (int i = 0; i < gangCount; i++)
        {
            gangs[i].strength = territoryControl[i];
            gangs[i].isActive = activityLevels[i] > 0.5f;
        }

        territoryCenters.Dispose();
        gangStrengths.Dispose();
        territoryControl.Dispose();
        activityLevels.Dispose();
    }

    MaldivianGang GenerateMaldivianGang(int index)
    {
        string[] territories = GetTerritoryList();
        MaldivianGang.GangType[] types = (MaldivianGang.GangType[])Syst
em.Enum.GetValues(typeof(MaldivianGang.GangType));

        MaldivianGang.GangType randomType = types[index % types.Length]
;
        string territory = territories[index % territories.Length];

        return new MaldivianGang
        {
            name = GenerateGangName(index, randomType),
            dhivehiName = GenerateDhivehiGangName(index),
            territory = territory,
            memberCount = UnityEngine.Random.Range(5, 50),
```

```
            type = randomType,
            strength = UnityEngine.Random.Range(0.1f, 1.0f),
            gangColor = new Color(UnityEngine.Random.Range(0f, 1f), Uni
tyEngine.Random.Range(0f, 1f), UnityEngine.Random.Range(0f, 1f)),
            isActive = true,
            activities = GenerateGangActivities(randomType)
        };
    }

    string GenerateGangName(int index, MaldivianGang.GangType type)
    {
        string[] prefixes = { "Black", "Red", "White", "Golden", "Silve
r", "Crimson", "Dark", "Shadow", "Night", "Blood" };
        string[] suffixes = { "Sharks", "Eagles", "Dragons", "Tigers",
"Wolves", "Serpents", "Scorpions", "Panthers", "Vipers", "Reapers" };

        return $"{prefixes[index % prefixes.Length]} {suffixes[index %
suffixes.Length]}";
    }

    string GenerateDhivehiGangName(int index)
    {
        string[] names = { "ކަސްޓަމަ ގެންޓް", "ދުވެރި ފުންބަ", "ތަރި ބައްދަލު", "ކަޅު ބައްދަލު",
 "ތަކި ތަކުރު" };
        return names[index % names.Length];
    }

    string[] GetTerritoryList()
    {
        return new string[]
        {
            "Malé North", "Malé South", "Malé Central", "Hulhumalé", "V
illimalé",
            "Addu City", "Hithadhoo", "Maradhoo", "Feydhoo", "Hulhudhoo
",
            "Naifaru", "Kulhudhuffushi", "Eydhafushi", "Ungoofaaru", "F
unadhoo",
            "Komandoo", "Veymandoo", "Muli", "Naifaru", "Kandoodhoo",
            "Thulusdhoo", "Himmafushi", "Huraa", "Maafushi", "Gulhi",
            "Guraidhoo", "Fulidhoo", "Keyodhoo", "Rakeedhoo", "Thinadho
o",
            "Kudahuvadhoo", "Veymandoo", "Maaenboodhoo", "Bileydhoo", "
Gadhdhoo",
            "Fonadhoo", "Dhanbidhoo", "Maamendhoo", "Kunahandhoo", "Dhi
yamingili",
            "Madaveli", "Hoandeddhoo", "Rathafandhoo", "Vaadhoo", "Kola
maafushi",
            "Kanduhulhudhoo", "Nilandhoo", "Dhadimago", "Biledhdhoo", "
Dhoondigan",
```

```
            "Faresmaathodaa", "Magoodhoo", "Dhevvadhoo", "Kondeymatheel
a", "Dhiyaree",
            "Fiyoaree", "Maamendhoo", "Dhevva", "Fiyoaree", "Kanduvale"
,
            "Bodufolhudhoo", "Feridhoo", "Mathiveri", "Bathalaa", "Kuda
folhudhoo",
            "Maalhos", "Rasdhoo", "Ukulhas", "Mathiveri", "Feridhoo",
            "Bodufolhudhoo", "Himandhoo", "Thoddoo", "Feridhoo", "Maalh
os",
            "Rasdhoo", "Ukulhas", "Mathiveri", "Bathalaa", "Kudafolhudh
oo",
            "Maamigili", "Dhigurah", "Fenfushi", "Mahibadhoo", "Dhanget
hi"
        };
    }

    float3 GetTerritoryCenter(string territory)
    {
        // Simplified territory positioning
        int hash = territory.GetHashCode();
        float lat = 2f + (hash % 100) * 0.05f;
        float lng = 72f + (hash / 100) * 0.2f;
        return new float3(lat, 0, lng);
    }

    string[] GenerateGangActivities(MaldivianGang.GangType type)
    {
        return type switch
        {
            MaldivianGang.GangType.StreetGang => new string[] { "Turf p
rotection", "Intimidation", "Petty theft" },
            MaldivianGang.GangType.DrugCartel => new string[] { "Drug t
rafficking", "Money laundering", "Corruption" },
            MaldivianGang.GangType.SmugglingRing => new string[] { "Goo
ds smuggling", "Human trafficking", "Tax evasion" },
            MaldivianGang.GangType.ProtectionRacket => new string[] { "
Extortion", "Security services", "Threats" },
            _ => new string[] { "Illegal activities", "Territory disput
es", "Criminal operations" }
        };
    }
}
```

12. BuildingSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
```

```
using Unity.Mathematics;

[BurstCompile]
public class BuildingSystem : MonoBehaviour
{
    [BurstCompile]
    struct BuildingPlacement : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> terrainPoints;
        [ReadOnly] public NativeArray<float> terrainHeight;
        [WriteOnly] public NativeArray<bool> buildableArea;
        [WriteOnly] public NativeArray<int> buildingType;

        public float maxSlope;
        public float minHeight;
        public float maxHeight;

        public void Execute(int index)
        {
            float3 point = terrainPoints[index];
            float height = terrainHeight[index];

            // Determine if area is suitable for building
            buildableArea[index] = IsBuildable(point, height);
            buildingType[index] = DetermineBuildingType(point, height);
        }

        [BurstCompile]
        bool IsBuildable(float3 point, float height)
        {
            // Check height constraints
            if (height < minHeight || height > maxHeight)
                return false;

            // Check slope (simplified)
            float slope = CalculateSlope(point);
            return slope < maxSlope;
        }

        [BurstCompile]
        float CalculateSlope(float3 point)
        {
            // Simplified slope calculation
            float heightVariation = math.length(new float3(
                math.sin(point.x * 10) * 0.1f,
                0,
                math.cos(point.z * 10) * 0.1f
            ));
```

```
            return heightVariation;
        }

        [BurstCompile]
        int DetermineBuildingType(float3 point, float height)
        {
            // Maldivian building types based on location
            if (height < 1f) // Coastal
                return 1; // House
            else if (height < 3f) // Residential
                return 2; // Apartment
            else if (height < 5f) // Commercial
                return 3; // Shop
            else
                return 4; // Mosque
        }
    }

    [BurstCompile]
    struct MaldivianArchitectureStyle : IJob
    {
        public NativeArray<float3> buildingColors;
        public NativeArray<float> buildingHeights;
        public int buildingCount;
        public float seed;

        public void Execute()
        {
            GenerateMaldivianStyle();
        }

        [BurstCompile]
        void GenerateMaldivianStyle()
        {
            // Traditional Maldivian architecture colors
            float3[] traditionalColors = new float3[]
            {
                new float3(0.9f, 0.8f, 0.6f), // Coral white
                new float3(0.8f, 0.6f, 0.4f), // Coral brown
                new float3(0.6f, 0.8f, 0.9f), // Ocean blue
                new float3(0.9f, 0.9f, 0.7f), // Sand yellow
                new float3(0.7f, 0.7f, 0.9f)  // Sky blue
            };

            for (int i = 0; i < buildingCount; i++)
            {
                buildingColors[i] = traditionalColors[i % traditionalCo
lors.Length];
                buildingHeights[i] = 3.0f + (i % 5) * 2.0f; // Varying
```

```
heights
            }
        }
    }

    public static BuildingSystem Instance { get; private set; }

    [System.Serializable]
    public class MaldivianBuilding
    {
        public string name;
        public string dhivehiName;
        public BuildingType type;
        public Vector3 position;
        public float height;
        public Color color;
        public bool isHistorical;
        public string[] functions;
        public int floorCount;

        public enum BuildingType
        {
            House,
            Apartment,
            Shop,
            Mosque,
            School,
            Hospital,
            GovernmentOffice,
            Hotel,
            Restaurant,
            Warehouse,
            Harbor,
            Market,
            Bank,
            PoliceStation,
            FireStation,
            Temple, // Historical
            Fort, // Historical
            TraditionalHouse,
            CoralHouse,
            ModernVilla
        }
    }

    public MaldivianBuilding[] buildings; // 70 buildings

    void Awake()
    {
```

```
        Instance = this;
        GenerateBuildingSystem();
    }

    void GenerateBuildingSystem()
    {
        const int buildingCount = 70; // Total buildings
        buildings = new MaldivianBuilding[buildingCount];

        var terrainPoints = new NativeArray<float3>(buildingCount, Allo
cator.TempJob);
        var terrainHeights = new NativeArray<float>(buildingCount, Allo
cator.TempJob);
        var buildableAreas = new NativeArray<bool>(buildingCount, Alloc
ator.TempJob);
        var buildingTypes = new NativeArray<int>(buildingCount, Allocat
or.TempJob);
        var buildingColors = new NativeArray<float3>(buildingCount, All
ocator.TempJob);
        var buildingHeights = new NativeArray<float>(buildingCount, All
ocator.TempJob);

        // Generate terrain data
        for (int i = 0; i < buildingCount; i++)
        {
            float lat = UnityEngine.Random.Range(2f, 7f);
            float lng = UnityEngine.Random.Range(72f, 74f);
            terrainPoints[i] = new float3(lat, 0, lng);
            terrainHeights[i] = GetTerrainHeight(lat, lng);
        }

        var placementJob = new BuildingPlacement
        {
            terrainPoints = terrainPoints,
            terrainHeight = terrainHeights,
            buildableArea = buildableAreas,
            buildingType = buildingTypes,
            maxSlope = 0.3f,
            minHeight = 0.5f,
            maxHeight = 10f
        };

        var styleJob = new MaldivianArchitectureStyle
        {
            buildingColors = buildingColors,
            buildingHeights = buildingHeights,
            buildingCount = buildingCount,
            seed = Time.time
        };
```

```
        JobHandle placementHandle = placementJob.Schedule(buildingCount
, 16);
        JobHandle styleHandle = styleJob.Schedule(placementHandle);
        styleHandle.Complete();

        // Create buildings
        for (int i = 0; i < buildingCount; i++)
        {
            if (buildableAreas[i])
            {
                buildings[i] = CreateBuilding(i, terrainPoints[i], buil
dingTypes[i], buildingColors[i], buildingHeights[i]);
            }
        }

        terrainPoints.Dispose();
        terrainHeights.Dispose();
        buildableAreas.Dispose();
        buildingTypes.Dispose();
        buildingColors.Dispose();
        buildingHeights.Dispose();
    }

    MaldivianBuilding CreateBuilding(int index, float3 position, int ty
pe, float3 color, float height)
    {
        MaldivianBuilding.BuildingType buildingType = (MaldivianBuildin
g.BuildingType)type;

        return new MaldivianBuilding
        {
            name = GenerateBuildingName(index, buildingType),
            dhivehiName = GenerateDhivehiBuildingName(buildingType),
            type = buildingType,
            position = position,
            height = height,
            color = new Color(color.x, color.y, color.z),
            isHistorical = IsHistoricalBuilding(buildingType),
            functions = GetBuildingFunctions(buildingType),
            floorCount = Mathf.FloorToInt(height / 3f)
        };
    }

    string GenerateBuildingName(int index, MaldivianBuilding.BuildingTy
pe type)
    {
        return $"{type}_{index}";
    }
```

```
    string GenerateDhivehiBuildingName(MaldivianBuilding.BuildingType t
ype)
    {
        return type switch
        {
            MaldivianBuilding.BuildingType.House => "ގެ",
            MaldivianBuilding.BuildingType.Apartment => "ފްލެޓްސް",
            MaldivianBuilding.BuildingType.Shop => "ހަލްއަޅާ",
            MaldivianBuilding.BuildingType.Mosque => "މަސްޖިދު",
            MaldivianBuilding.BuildingType.School => "ދަރަނޑު",
            MaldivianBuilding.BuildingType.Hospital => "ދާދިފަރު",
            MaldivianBuilding.BuildingType.GovernmentOffice => "ސަރުކާ
ރީ",
            MaldivianBuilding.BuildingType.Hotel => "ހޯޓަލް",
            MaldivianBuilding.BuildingType.Restaurant => "ތަނަވަސްތަ",
            _ => "ބިލްޑިންގ"
        };
    }

    bool IsHistoricalBuilding(MaldivianBuilding.BuildingType type)
    {
        return type == MaldivianBuilding.BuildingType.Temple ||
               type == MaldivianBuilding.BuildingType.Fort ||
               type == MaldivianBuilding.BuildingType.TraditionalHouse
||
               type == MaldivianBuilding.BuildingType.CoralHouse;
    }

    string[] GetBuildingFunctions(MaldivianBuilding.BuildingType type)
    {
        return type switch
        {
            MaldivianBuilding.BuildingType.House => new string[] { "Res
idential", "Family living" },
            MaldivianBuilding.BuildingType.Shop => new string[] { "Comm
ercial", "Retail", "Services" },
            MaldivianBuilding.BuildingType.Mosque => new string[] { "Re
ligious", "Community", "Education" },
            MaldivianBuilding.BuildingType.School => new string[] { "Ed
ucation", "Learning", "Childcare" },
            MaldivianBuilding.BuildingType.Hospital => new string[] { "
Healthcare", "Emergency", "Treatment" },
            _ => new string[] { "Building", "Structure" }
        };
```

```
    }

    float GetTerrainHeight(float latitude, float longitude)
    {
        // Simplified height map based on island locations
        float islandNoise = Mathf.PerlinNoise(latitude * 10, longitude
* 10);
        return islandNoise * 5f;
    }
}
```

13. MissionSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class MissionSystem : MonoBehaviour
{
    [BurstCompile]
    struct MissionProgression : IJobParallelFor
    {
        [ReadOnly] public NativeArray<int> missionStates;
        [ReadOnly] public NativeArray<float> completionProgress;
        [WriteOnly] public NativeArray<bool> missionComplete;
        [WriteOnly] public NativeArray<int> nextMissionIDs;

        public void Execute(int index)
        {
            int state = missionStates[index];
            float progress = completionProgress[index];

            missionComplete[index] = progress >= 1.0f;
            nextMissionIDs[index] = missionComplete[index] ? state + 1
: -1;
        }
    }

    [BurstCompile]
    struct MissionRewardCalculation : IJob
    {
        public NativeArray<float> rewards;
        public NativeArray<int> missionTypes;
        public NativeArray<int> difficultyLevels;
```

```
        public void Execute()
        {
            for (int i = 0; i < rewards.Length; i++)
            {
                rewards[i] = CalculateReward(missionTypes[i], difficult
yLevels[i]);
            }
        }

        [BurstCompile]
        float CalculateReward(int missionType, int difficulty)
        {
            float baseReward = missionType switch
            {
                1 => 100f, // Story mission
                2 => 50f,  // Side mission
                3 => 75f,  // Gang mission
                4 => 30f,  // Delivery mission
                5 => 200f, // Assassination
                _ => 25f
            };

            float difficultyMultiplier = difficulty switch
            {
                1 => 0.5f,
                2 => 1.0f,
                3 => 1.5f,
                4 => 2.0f,
                5 => 3.0f,
                _ => 1.0f
            };

            return baseReward * difficultyMultiplier;
        }
    }

    public static MissionSystem Instance { get; private set; }

    [System.Serializable]
    public class MaldivianMission
    {
        public int missionID;
        public string missionName;
        public string dhivehiName;
        public string description;
        public MissionType type;
        public DifficultyLevel difficulty;
        public Vector3 startLocation;
        public Vector3 targetLocation;
```

```
        public string[] objectives;
        public float[] objectiveProgress;
        public float reward;
        public bool isComplete;
        public bool isActive;
        public int[] prerequisiteMissions;
        public string[] dialogueSequences;
        public GameObject[] relatedNPCs;
        public float timeLimit;

        public enum MissionType
        {
            Story,
            Side,
            Gang,
            Delivery,
            Assassination,
            Collection,
            Protection,
            Racing,
            Fishing,
            Trading,
            Religious,
            Cultural,
            Political,
            Emergency,
            RandomEvent
        }

        public enum DifficultyLevel { Easy = 1, Normal = 2, Hard = 3, E
xpert = 4, Master = 5 }
    }

    public MaldivianMission[] missions; // 100+ missions

    void Awake()
    {
        Instance = this;
        InitializeMissionSystem();
    }

    void InitializeMissionSystem()
    {
        const int missionCount = 150; // Total missions
        missions = new MaldivianMission[missionCount];

        var missionStates = new NativeArray<int>(missionCount, Allocato
r.TempJob);
        var completionProgress = new NativeArray<float>(missionCount, A
```

```
llocator.TempJob);
        var missionComplete = new NativeArray<bool>(missionCount, Alloc
ator.TempJob);
        var nextMissionIDs = new NativeArray<int>(missionCount, Allocat
or.TempJob);
        var rewards = new NativeArray<float>(missionCount, Allocator.Te
mpJob);
        var missionTypes = new NativeArray<int>(missionCount, Allocator
.TempJob);
        var difficultyLevels = new NativeArray<int>(missionCount, Alloc
ator.TempJob);

        // Initialize missions
        for (int i = 0; i < missionCount; i++)
        {
            missions[i] = GenerateMaldivianMission(i);
            missionStates[i] = (int)missions[i].type;
            completionProgress[i] = 0f;
            missionTypes[i] = (int)missions[i].type;
            difficultyLevels[i] = (int)missions[i].difficulty;
        }

        var progressionJob = new MissionProgression
        {
            missionStates = missionStates,
            completionProgress = completionProgress,
            missionComplete = missionComplete,
            nextMissionIDs = nextMissionIDs
        };

        var rewardJob = new MissionRewardCalculation
        {
            rewards = rewards,
            missionTypes = missionTypes,
            difficultyLevels = difficultyLevels
        };

        JobHandle progressionHandle = progressionJob.Schedule(missionCo
unt, 16);
        JobHandle rewardHandle = rewardJob.Schedule(progressionHandle);
        rewardHandle.Complete();

        // Update mission rewards
        for (int i = 0; i < missionCount; i++)
        {
            missions[i].reward = rewards[i];
        }

        missionStates.Dispose();
```

```
        completionProgress.Dispose();
        missionComplete.Dispose();
        nextMissionIDs.Dispose();
        rewards.Dispose();
        missionTypes.Dispose();
        difficultyLevels.Dispose();
    }

    MaldivianMission GenerateMaldivianMission(int index)
    {
        MaldivianMission.MissionType[] types = (MaldivianMission.Missio
nType[])System.Enum.GetValues(typeof(MaldivianMission.MissionType));
        MaldivianMission.MissionType randomType = types[index % types.L
ength];

        return new MaldivianMission
        {
            missionID = index,
            missionName = GenerateMissionName(index, randomType),
            dhivehiName = GenerateDhivehiMissionName(randomType),
            description = GenerateMissionDescription(randomType),
            type = randomType,
            difficulty = (MaldivianMission.DifficultyLevel)(index % 5 +
 1),
            startLocation = GetRandomLocation(),
            targetLocation = GetRandomLocation(),
            objectives = GenerateObjectives(randomType),
            objectiveProgress = new float[3],
            reward = 0, // Calculated by job
            isComplete = false,
            isActive = index == 0, // First mission active
            prerequisiteMissions = index > 0 ? new int[] { index - 1 }
: new int[0],
            dialogueSequences = GenerateDialogue(randomType),
            relatedNPCs = new GameObject[0],
            timeLimit = GetTimeLimit(randomType)
        };
    }

    string GenerateMissionName(int index, MaldivianMission.MissionType
type)
    {
        return type switch
        {
            MaldivianMission.MissionType.Story => $"The Albako Chronicl
es - Chapter {index}",
            MaldivianMission.MissionType.Side => $"Island Task {index}"
,
            MaldivianMission.MissionType.Gang => $"Turf War {index}",
```

```
            MaldivianMission.MissionType.Delivery => $"Delivery Job {in
dex}",
            MaldivianMission.MissionType.Assassination => $"Target Elim
ination {index}",
            MaldivianMission.MissionType.Collection => $"Resource Gathe
ring {index}",
            MaldivianMission.MissionType.Protection => $"Security Detai
l {index}",
            MaldivianMission.MissionType.Racing => $"Island Race {index
}",
            MaldivianMission.MissionType.Fishing => $"Traditional Fishi
ng {index}",
            MaldivianMission.MissionType.Trading => $"Business Deal {in
dex}",
            MaldivianMission.MissionType.Religious => $"Spiritual Journ
ey {index}",
            MaldivianMission.MissionType.Cultural => $"Cultural Heritag
e {index}",
            MaldivianMission.MissionType.Political => $"Political Intri
gue {index}",
            MaldivianMission.MissionType.Emergency => $"Crisis Response
 {index}",
            MaldivianMission.MissionType.RandomEvent => $"Unexpected Ev
ent {index}",
            _ => $"Mission {index}"
        };
    }

    string GenerateDhivehiMissionName(MaldivianMission.MissionType type
)
    {
        return type switch
        {
            MaldivianMission.MissionType.Story => "ތާރީހުގެ ކުރިއަށްގެންދާ",
            MaldivianMission.MissionType.Side => "ފަރާތުގެ ހަދިޔާ",
            MaldivianMission.MissionType.Gang => "ގެންދާ ބަންދުގެ",
            MaldivianMission.MissionType.Delivery => "ގެނެވޭ ހަދިޔާ",
            MaldivianMission.MissionType.Assassination => "ކަނޑާ ދުރުން",
            MaldivianMission.MissionType.Collection => "ގުޅައިގެން ދާ ތަން",
            MaldivianMission.MissionType.Protection => "ތަހައްކުދާ",
            MaldivianMission.MissionType.Racing => "ދާ ދާ",
            MaldivianMission.MissionType.Fishing => "ހަދާ ފުށުން",
            MaldivianMission.MissionType.Trading => "ވަޒަނ",
            MaldivianMission.MissionType.Religious => "ދީނީ ހަދިޔާ",
            MaldivianMission.MissionType.Cultural => "ތަރައްޤީ ހަދިޔާ",
```

```
            MaldivianMission.MissionType.Political => "ﷺ ﷺ",
            MaldivianMission.MissionType.Emergency => "ﷺ",
            MaldivianMission.MissionType.RandomEvent => "ﷺ",
            _ => "ﷺ"
        };
    }

    string GenerateMissionDescription(MaldivianMission.MissionType type
)
    {
        return type switch
        {
            MaldivianMission.MissionType.Story => "Uncover the secrets
of the Albako Chronicles and restore balance to the islands.",
            MaldivianMission.MissionType.Side => "Help local islanders
with their daily challenges and earn their trust.",
            MaldivianMission.MissionType.Gang => "Navigate the dangerou
s world of island gangs and establish your influence.",
            MaldivianMission.MissionType.Delivery => "Transport goods b
etween islands while avoiding pirates and customs.",
            MaldivianMission.MissionType.Assassination => "Eliminate hi
gh-value targets for powerful clients.",
            MaldivianMission.MissionType.Collection => "Gather rare res
ources from across the archipelago.",
            MaldivianMission.MissionType.Protection => "Guard important
 cargo or individuals from threats.",
            MaldivianMission.MissionType.Racing => "Compete in traditio
nal boat races and modern vehicle challenges.",
            MaldivianMission.MissionType.Fishing => "Participate in tra
ditional Maldivian fishing practices.",
            MaldivianMission.MissionType.Trading => "Buy and sell goods
 across the island economy.",
            MaldivianMission.MissionType.Religious => "Participate in I
slamic traditions and spiritual practices.",
            MaldivianMission.MissionType.Cultural => "Preserve and cele
brate Maldivian cultural heritage.",
            MaldivianMission.MissionType.Political => "Navigate the com
plex political landscape of the islands.",
            MaldivianMission.MissionType.Emergency => "Respond to urgen
t situations and save lives.",
            MaldivianMission.MissionType.RandomEvent => "Deal with unex
pected situations as they arise.",
            _ => "Complete the assigned task."
        };
    }

    string[] GenerateObjectives(MaldivianMission.MissionType type)
    {
```

```
        return type switch
        {
            MaldivianMission.MissionType.Story => new string[] { "Inves
tigate the mystery", "Gather clues", "Confront the truth" },
            MaldivianMission.MissionType.Delivery => new string[] { "Pi
ck up package", "Deliver to destination", "Avoid detection" },
            MaldivianMission.MissionType.Assassination => new string[]
{ "Locate target", "Plan approach", "Eliminate target" },
            MaldivianMission.MissionType.Collection => new string[] { "
Find items", "Collect resources", "Return to client" },
            _ => new string[] { "Complete main objective", "Avoid compl
ications", "Report back" }
        };
    }

    Vector3 GetRandomLocation()
    {
        return new Vector3(UnityEngine.Random.Range(2f, 7f), 0, UnityEn
gine.Random.Range(72f, 74f));
    }

    string[] GenerateDialogue(MaldivianMission.MissionType type)
    {
        return new string[] { "Mission briefing", "Mid-mission update",
 "Mission completion" };
    }

    float GetTimeLimit(MaldivianMission.MissionType type)
    {
        return type switch
        {
            MaldivianMission.MissionType.Emergency => 300f, // 5 minute
s
            MaldivianMission.MissionType.Racing => 600f,    // 10 minut
es
            MaldivianMission.MissionType.Delivery => 1800f, // 30 minut
es
            _ => 3600f // 1 hour default
        };
    }
}
```

14. DialogueSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
```

```
[BurstCompile]
public class DialogueSystem : MonoBehaviour
{
    [BurstCompile]
    struct DialogueBranching : IJobParallelFor
    {
        [ReadOnly] public NativeArray<int> dialogueStates;
        [ReadOnly] public NativeArray<int> playerChoices;
        [WriteOnly] public NativeArray<int> nextDialogueNodes;
        [WriteOnly] public NativeArray<float> relationshipChanges;

        public void Execute(int index)
        {
            int state = dialogueStates[index];
            int choice = playerChoices[index];

            // Calculate dialogue progression
            nextDialogueNodes[index] = CalculateNextNode(state, choice)
;
            relationshipChanges[index] = CalculateRelationshipChange(st
ate, choice);
        }

        [BurstCompile]
        int CalculateNextNode(int currentState, int choice)
        {
            // Branching logic based on choice
            return currentState * 10 + choice + 1;
        }

        [BurstCompile]
        float CalculateRelationshipChange(int state, int choice)
        {
            // Positive choices increase relationship
            return choice == 0 ? 0.1f : choice == 1 ? 0.05f : -0.05f;
        }
    }

    [BurstCompile]
    struct DhivehiLanguageProcessing : IJob
    {
        public NativeArray<char> inputText;
        public NativeArray<char> outputText;

        public void Execute()
        {
            // Process Dhivehi text for display
            ProcessDhivehiText();
```

```
        }

        [BurstCompile]
        void ProcessDhivehiText()
        {
            // Simplified text processing
            for (int i = 0; i < inputText.Length && i < outputText.Leng
th; i++)
            {
                outputText[i] = inputText[i];
            }
        }
    }

    public static DialogueSystem Instance { get; private set; }

    [System.Serializable]
    public class DialogueNode
    {
        public int nodeID;
        public string speakerName;
        public string dhivehiSpeakerName;
        public string text;
        public string dhivehiText;
        public DialogueChoice[] choices;
        public string[] conditions;
        public string[] consequences;
        public float duration;
        public AudioClip voiceLine;
        public bool isImportant;
        public string emotion;
        public string animationTrigger;
    }

    [System.Serializable]
    public class DialogueChoice
    {
        public int choiceID;
        public string text;
        public string dhivehiText;
        public int nextNodeID;
        public string[] requirements;
        public string[] effects;
        public float relationshipImpact;
        public bool endsDialogue;
    }

    public DialogueNode[] dialogueDatabase; // 500+ dialogue nodes
```

```
    void Awake()
    {
        Instance = this;
        InitializeDialogueSystem();
    }

    void InitializeDialogueSystem()
    {
        const int dialogueCount = 500; // Total dialogue nodes
        dialogueDatabase = new DialogueNode[dialogueCount];

        var dialogueStates = new NativeArray<int>(dialogueCount, Alloca
tor.TempJob);
        var playerChoices = new NativeArray<int>(dialogueCount, Allocat
or.TempJob);
        var nextNodes = new NativeArray<int>(dialogueCount, Allocator.T
empJob);
        var relationshipChanges = new NativeArray<float>(dialogueCount,
 Allocator.TempJob);

        // Initialize dialogue nodes
        for (int i = 0; i < dialogueCount; i++)
        {
            dialogueDatabase[i] = GenerateDialogueNode(i);
            dialogueStates[i] = i;
            playerChoices[i] = UnityEngine.Random.Range(0, 3);
        }

        var branchingJob = new DialogueBranching
        {
            dialogueStates = dialogueStates,
            playerChoices = playerChoices,
            nextDialogueNodes = nextNodes,
            relationshipChanges = relationshipChanges
        };

        var dhivehiJob = new DhivehiLanguageProcessing
        {
            inputText = new NativeArray<char>(100, Allocator.TempJob),
            outputText = new NativeArray<char>(100, Allocator.TempJob)
        };

        JobHandle branchingHandle = branchingJob.Schedule(dialogueCount
, 32);
        JobHandle dhivehiHandle = dhivehiJob.Schedule(branchingHandle);
        dhivehiHandle.Complete();

        dialogueStates.Dispose();
        playerChoices.Dispose();
```

```
        nextNodes.Dispose();
        relationshipChanges.Dispose();
    }

    DialogueNode GenerateDialogueNode(int index)
    {
        string[] speakers = { "Villager", "Fisherman", "Shopkeeper", "E
lder", "Youth", "Official", "Tourist", "ReligiousLeader" };
        string speaker = speakers[index % speakers.Length];

        return new DialogueNode
        {
            nodeID = index,
            speakerName = speaker,
            dhivehiSpeakerName = GetDhivehiSpeakerName(speaker),
            text = GenerateDialogueText(index, speaker),
            dhivehiText = GenerateDhivehiDialogue(index),
            choices = GenerateDialogueChoices(index),
            conditions = new string[0],
            consequences = new string[0],
            duration = 3.0f,
            voiceLine = null,
            isImportant = index % 10 == 0,
            emotion = GetRandomEmotion(),
            animationTrigger = GetAnimationTrigger(index)
        };
    }

    string GetDhivehiSpeakerName(string englishName)
    {
        return englishName switch
        {
            "Villager" => "ތަރިއްޔާ",
            "Fisherman" => "ދަންވެރިން",
            "Shopkeeper" => "ހަވާލުވެރި",
            "Elder" => "ބުޒުރުން",
            "Youth" => "ފަތަރާ",
            "Official" => "ސަރުކާރުގެ ބުރަދަރު",
            "Tourist" => "ފާތިހު",
            "ReligiousLeader" => "ދީނީ ތަރުބިޔަތްދޭ",
            _ => "ރާއްޖެ"
        };
    }

    string GenerateDialogueText(int index, string speaker)
    {
```

```
    string[] templates = {
        "Welcome to our island, traveler.",
        "The sea has been rough lately.",
        "Have you heard about the recent events?",
        "Life here is simple but meaningful.",
        "We follow the old traditions here.",
        "The fishing has been good this season.",
        "Peace be upon you.",
        "How can I help you today?"
    };

    return templates[index % templates.Length];
}

string GenerateDhivehiDialogue(int index)
{
    string[] templates = {
        "ދަތުރުގައި، ދެބަހުދަތްކޮ ތަކެއްގެ.",
        "ތިޔަ ދުވަސް ގޮތްތަކުގެ.",
        "ތިޔަކަށް ހަގީގީ ތަކެއް؟",
        "ފެންނަ ގޮތަށް ސާދަ ނަމަވެސް ތަކެއް.",
        "ތަކުރާރުވާ ގޮތަށް ކުރިއަށް.",
        "ދަތުރު ކުރާ ތަނެއް.",
        "ތަކެއް ކަމަށް ކަމަށް.",
        "ތަކެއް ކަމަށް ތިބެ ތަކެއް؟"
    };

    return templates[index % templates.Length];
}

DialogueChoice[] GenerateDialogueChoices(int index)
{
    return new DialogueChoice[]
    {
        new DialogueChoice
        {
            choiceID = 0,
            text = "Tell me more.",
            dhivehiText = "ތަކެއް ވެސް.",
            nextNodeID = index + 1,
            requirements = new string[0],
            effects = new string[0],
            relationshipImpact = 0.1f,
            endsDialogue = false
        },
        new DialogueChoice
```

```
            {
                choiceID = 1,
                text = "Goodbye.",
                dhivehiText = "ފުރަހަށްދާ.",
                nextNodeID = -1,
                requirements = new string[0],
                effects = new string[0],
                relationshipImpact = 0f,
                endsDialogue = true
            }
        };
    }

    string GetRandomEmotion()
    {
        string[] emotions = { "neutral", "happy", "sad", "angry", "surp
rised", "confused" };
        return emotions[UnityEngine.Random.Range(0, emotions.Length)];
    }

    string GetAnimationTrigger(int index)
    {
        return index % 5 == 0 ? "talk_important" : "talk_normal";
    }
}
```

15. InventorySystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class InventorySystem : MonoBehaviour
{
    [BurstCompile]
    struct InventoryOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> itemSizes;
        [ReadOnly] public NativeArray<float> itemWeights;
        [WriteOnly] public NativeArray<int> optimalSlots;

        public float maxWeight;
        public int maxSlots;

        public void Execute(int index)
```

```
        {
            float size = itemSizes[index];
            float weight = itemWeights[index];

            // Optimize inventory placement
            optimalSlots[index] = CalculateOptimalSlot(size, weight, in
dex);
        }

        [BurstCompile]
        int CalculateOptimalSlot(float size, float weight, int itemInde
x)
        {
            // Simple slot assignment based on size and weight
            if (weight > maxWeight * 0.7f)
                return 0; // Heavy items in first slots
            else if (size > 0.5f)
                return 1; // Large items in second slot group
            else
                return 2 + (itemIndex % (maxSlots - 2)); // Small items
 distributed
        }
    }

    [BurstCompile]
    struct MaldivianItemGeneration : IJob
    {
        public NativeArray<float> itemValues;
        public NativeArray<int> itemCategories;
        public NativeArray<bool> isTraditionalItems;

        public void Execute()
        {
            GenerateMaldivianItems();
        }

        [BurstCompile]
        void GenerateMaldivianItems()
        {
            for (int i = 0; i < itemValues.Length; i++)
            {
                // Traditional Maldivian items have higher cultural val
ue
                bool isTraditional = i % 3 == 0;
                isTraditionalItems[i] = isTraditional;
                itemValues[i] = isTraditional ? 2.0f : 1.0f;
                itemCategories[i] = i % 10;
            }
        }
```

```
    }

    public static InventorySystem Instance { get; private set; }

    [System.Serializable]
    public class InventoryItem
    {
        public int itemID;
        public string itemName;
        public string dhivehiName;
        public string description;
        public ItemCategory category;
        public float weight;
        public float size;
        public int stackSize;
        public int currentStack;
        public float value;
        public bool isTradable;
        public bool isQuestItem;
        public bool isTraditional;
        public GameObject itemModel;
        public Sprite itemIcon;
        public string[] properties;

        public enum ItemCategory
        {
            Weapon,
            Tool,
            Food,
            Material,
            Treasure,
            Clothing,
            Electronic,
            Document,
            Religious,
            Cultural,
            Medical,
            Crafting,
            Consumable,
            Quest,
            Illegal
        }
    }

    public InventoryItem[] inventoryItems; // 200+ items
    public InventoryItem[] playerInventory; // Player's current invento
ry

    void Awake()
```

```
    {
        Instance = this;
        InitializeInventorySystem();
    }

    void InitializeInventorySystem()
    {
        const int totalItems = 200; // Total unique items
        const int inventorySize = 50; // Player inventory size

        inventoryItems = new InventoryItem[totalItems];
        playerInventory = new InventoryItem[inventorySize];

        var itemSizes = new NativeArray<float>(totalItems, Allocator.Te
mpJob);
        var itemWeights = new NativeArray<float>(totalItems, Allocator.
TempJob);
        var optimalSlots = new NativeArray<int>(totalItems, Allocator.T
empJob);
        var itemValues = new NativeArray<float>(totalItems, Allocator.T
empJob);
        var itemCategories = new NativeArray<int>(totalItems, Allocator
.TempJob);
        var isTraditionalItems = new NativeArray<bool>(totalItems, Allo
cator.TempJob);

        // Initialize items
        for (int i = 0; i < totalItems; i++)
        {
            inventoryItems[i] = GenerateMaldivianItem(i);
            itemSizes[i] = inventoryItems[i].size;
            itemWeights[i] = inventoryItems[i].weight;
            itemValues[i] = inventoryItems[i].value;
            itemCategories[i] = (int)inventoryItems[i].category;
        }

        var optimizationJob = new InventoryOptimization
        {
            itemSizes = itemSizes,
            itemWeights = itemWeights,
            optimalSlots = optimalSlots,
            maxWeight = 100f,
            maxSlots = inventorySize
        };

        var generationJob = new MaldivianItemGeneration
        {
            itemValues = itemValues,
            itemCategories = itemCategories,
```

```
            isTraditionalItems = isTraditionalItems
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(totalIt
ems, 32);
        JobHandle generationHandle = generationJob.Schedule(optimizatio
nHandle);
        generationHandle.Complete();

        // Update item properties
        for (int i = 0; i < totalItems; i++)
        {
            inventoryItems[i].isTraditional = isTraditionalItems[i];
        }

        itemSizes.Dispose();
        itemWeights.Dispose();
        optimalSlots.Dispose();
        itemValues.Dispose();
        itemCategories.Dispose();
        isTraditionalItems.Dispose();
    }

    InventoryItem GenerateMaldivianItem(int index)
    {
        InventoryItem.ItemCategory[] categories = (InventoryItem.ItemCa
tegory[])System.Enum.GetValues(typeof(InventoryItem.ItemCategory));
        InventoryItem.ItemCategory randomCategory = categories[index %
categories.Length];

        return new InventoryItem
        {
            itemID = index,
            itemName = GenerateItemName(index, randomCategory),
            dhivehiName = GenerateDhivehiItemName(randomCategory),
            description = GenerateItemDescription(randomCategory),
            category = randomCategory,
            weight = UnityEngine.Random.Range(0.1f, 5.0f),
            size = UnityEngine.Random.Range(0.1f, 2.0f),
            stackSize = GetStackSize(randomCategory),
            currentStack = 1,
            value = UnityEngine.Random.Range(10f, 1000f),
            isTradable = IsTradable(randomCategory),
            isQuestItem = index % 20 == 0,
            isTraditional = IsTraditional(randomCategory),
            itemModel = null,
            itemIcon = null,
            properties = GenerateItemProperties(randomCategory)
        };
```

```
    }

    string GenerateItemName(int index, InventoryItem.ItemCategory categ
ory)
    {
        return category switch
        {
            InventoryItem.ItemCategory.Weapon => $"Traditional Weapon {
index}",
            InventoryItem.ItemCategory.Tool => $"Island Tool {index}",
            InventoryItem.ItemCategory.Food => $"Maldivian Food {index}
",
            InventoryItem.ItemCategory.Material => $"Local Material {in
dex}",
            InventoryItem.ItemCategory.Treasure => $"Island Treasure {i
ndex}",
            InventoryItem.ItemCategory.Cultural => $"Cultural Artifact
{index}",
            InventoryItem.ItemCategory.Religious => $"Religious Item {i
ndex}",
            _ => $"Item {index}"
        };
    }

    string GenerateDhivehiItemName(InventoryItem.ItemCategory category)
    {
        return category switch
        {
            InventoryItem.ItemCategory.Weapon => "ހަތުފަދަ",
            InventoryItem.ItemCategory.Tool => "ތަކެއް",
            InventoryItem.ItemCategory.Food => "ދެރުވާ",
            InventoryItem.ItemCategory.Material => "ދަށެއް",
            InventoryItem.ItemCategory.Treasure => "ބަހައްޓާ",
            InventoryItem.ItemCategory.Cultural => "ތާރީހުގެ ތަކެއް",
            InventoryItem.ItemCategory.Religious => "ދީނީ ތަކެއް",
            _ => "ތަކެއް"
        };
    }

    string GenerateItemDescription(InventoryItem.ItemCategory category)
    {
        return category switch
        {
            InventoryItem.ItemCategory.Weapon => "A traditional Maldivi
an weapon used for protection.",
            InventoryItem.ItemCategory.Tool => "An essential tool for i
```

```
sland life and fishing.",
            InventoryItem.ItemCategory.Food => "Traditional Maldivian c
uisine with local ingredients.",
            InventoryItem.ItemCategory.Material => "Raw materials sourc
ed from the islands.",
            InventoryItem.ItemCategory.Treasure => "A valuable item wit
h cultural significance.",
            InventoryItem.ItemCategory.Cultural => "An artifact represe
nting Maldivian heritage.",
            InventoryItem.ItemCategory.Religious => "An item of religio
us importance in Islam.",
            _ => "A useful item found in the Maldives."
        };
    }

    int GetStackSize(InventoryItem.ItemCategory category)
    {
        return category switch
        {
            InventoryItem.ItemCategory.Weapon => 1,
            InventoryItem.ItemCategory.Tool => 1,
            InventoryItem.ItemCategory.Food => 20,
            InventoryItem.ItemCategory.Material => 100,
            InventoryItem.ItemCategory.Treasure => 1,
            InventoryItem.ItemCategory.Consumable => 50,
            _ => 10
        };
    }

    bool IsTradable(InventoryItem.ItemCategory category)
    {
        return category != InventoryItem.ItemCategory.Quest &&
               category != InventoryItem.ItemCategory.Illegal;
    }

    bool IsTraditional(InventoryItem.ItemCategory category)
    {
        return category == InventoryItem.ItemCategory.Cultural ||
               category == InventoryItem.ItemCategory.Religious ||
               category == InventoryItem.ItemCategory.Weapon;
    }

    string[] GenerateItemProperties(InventoryItem.ItemCategory category
)
    {
        return category switch
        {
            InventoryItem.ItemCategory.Weapon => new string[] { "Damage
: 10", "Durability: 100" },
```

```
            InventoryItem.ItemCategory.Tool => new string[] { "Efficien
cy: 80%", "Durability: 150" },
            InventoryItem.ItemCategory.Food => new string[] { "Nutritio
n: 25", "Freshness: 100%" },
            _ => new string[] { "Quality: Good", "Value: Standard" }
        };
    }
}
```

16. EconomySystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class EconomySystem : MonoBehaviour
{
    [BurstCompile]
    struct MarketSimulation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> basePrices;
        [ReadOnly] public NativeArray<float> supplyLevels;
        [ReadOnly] public NativeArray<float> demandLevels;
        [WriteOnly] public NativeArray<float> currentPrices;

        public float time;

        public void Execute(int index)
        {
            float basePrice = basePrices[index];
            float supply = supplyLevels[index];
            float demand = demandLevels[index];

            // Calculate market price based on supply and demand
            currentPrices[index] = CalculateMarketPrice(basePrice, supp
ly, demand);
        }

        [BurstCompile]
        float CalculateMarketPrice(float basePrice, float supply, float
 demand)
        {
            // Supply and demand mechanics
            float supplyDemandRatio = supply > 0 ? demand / supply : 1.
0f;
```

```
            float priceMultiplier = math.saturate(supplyDemandRatio);

            // Add some random fluctuation
            float fluctuation = math.sin(time + index) * 0.1f;

            return basePrice * (0.5f + priceMultiplier * 0.5f + fluctua
tion);
        }
    }

    [BurstCompile]
    struct MaldivianTradeRoutes : IJob
    {
        public NativeArray<float3> tradeCenters;
        public NativeArray<float> routeValues;

        public void Execute()
        {
            GenerateMaldivianTradeNetwork();
        }

        [BurstCompile]
        void GenerateMaldivianTradeNetwork()
        {
            // Real Maldivian trade centers
            float3[] centers = new float3[]
            {
                new float3(4.1755f, 0, 73.5093f), // Malé (main hub)
                new float3(4.7667f, 0, 73.3f),    // Addu
                new float3(5.2f, 0, 73.0f),       // Hithadhoo
                new float3(4.9f, 0, 73.3f),       // Naifaru
                new float3(5.5f, 0, 73.0f),       // Kulhudhuffushi
                new float3(4.6f, 0, 73.4f),       // Eydhafushi
                new float3(5.8f, 0, 73.4f),       // Ungoofaaru
                new float3(6.1f, 0, 73.3f)        // Funadhoo
            };

            for (int i = 0; i < centers.Length && i < tradeCenters.Leng
th; i++)
            {
                tradeCenters[i] = centers[i];
                routeValues[i] = 1000.0f + i * 100.0f; // Trade value d
ecreases with distance
            }
        }
    }

    public static EconomySystem Instance { get; private set; }
```

```
    [System.Serializable]
    public class EconomicItem
    {
        public int itemID;
        public string itemName;
        public string dhivehiName;
        public float basePrice;
        public float currentPrice;
        public float supply;
        public float demand;
        public bool isImported;
        public bool isExported;
        public float importTax;
        public float exportTax;
        public string[] relatedItems;
    }

    public EconomicItem[] marketItems; // 100+ tradeable items
    public float totalEconomyValue;
    public float dailyTradeVolume;

    void Awake()
    {
        Instance = this;
        InitializeEconomySystem();
    }

    void InitializeEconomySystem()
    {
        const int itemCount = 100; // Total economic items
        marketItems = new EconomicItem[itemCount];

        var basePrices = new NativeArray<float>(itemCount, Allocator.Te
mpJob);
        var supplyLevels = new NativeArray<float>(itemCount, Allocator.
TempJob);
        var demandLevels = new NativeArray<float>(itemCount, Allocator.
TempJob);
        var currentPrices = new NativeArray<float>(itemCount, Allocator
.TempJob);
        var tradeCenters = new NativeArray<float3>(10, Allocator.TempJo
b);
        var routeValues = new NativeArray<float>(10, Allocator.TempJob)
;

        // Initialize market items
        for (int i = 0; i < itemCount; i++)
        {
            marketItems[i] = GenerateEconomicItem(i);
```

```
            basePrices[i] = marketItems[i].basePrice;
            supplyLevels[i] = marketItems[i].supply;
            demandLevels[i] = marketItems[i].demand;
        }

        var marketJob = new MarketSimulation
        {
            basePrices = basePrices,
            supplyLevels = supplyLevels,
            demandLevels = demandLevels,
            currentPrices = currentPrices,
            time = Time.time
        };

        var tradeJob = new MaldivianTradeRoutes
        {
            tradeCenters = tradeCenters,
            routeValues = routeValues
        };

        JobHandle marketHandle = marketJob.Schedule(itemCount, 16);
        JobHandle tradeHandle = tradeJob.Schedule(marketHandle);
        tradeHandle.Complete();

        // Update current prices
        for (int i = 0; i < itemCount; i++)
        {
            marketItems[i].currentPrice = currentPrices[i];
        }

        basePrices.Dispose();
        supplyLevels.Dispose();
        demandLevels.Dispose();
        currentPrices.Dispose();
        tradeCenters.Dispose();
        routeValues.Dispose();

        CalculateEconomyMetrics();
    }

    EconomicItem GenerateEconomicItem(int index)
    {
        string[] items = { "Fish", "Coconuts", "Rice", "Sugar", "Textil
es", "Electronics", "Fuel", "Medicine", "ConstructionMaterials", "Touri
smServices" };
        string itemName = items[index % items.Length];

        return new EconomicItem
        {
```

```
            itemID = index,
            itemName = itemName,
            dhivehiName = GetDhivehiItemName(itemName),
            basePrice = UnityEngine.Random.Range(10f, 500f),
            currentPrice = 0, // Calculated by job
            supply = UnityEngine.Random.Range(0.1f, 1.0f),
            demand = UnityEngine.Random.Range(0.1f, 1.0f),
            isImported = index % 3 == 0,
            isExported = index % 4 == 0,
            importTax = index % 3 == 0 ? 0.15f : 0f,
            exportTax = index % 4 == 0 ? 0.1f : 0f,
            relatedItems = GenerateRelatedItems(itemName)
        };
    }

    string GetDhivehiItemName(string englishName)
    {
        return englishName switch
        {
            "Fish" => "ހަނޑު",
            "Coconuts" => "ފަނަރު",
            "Rice" => "ސަމާރު",
            "Sugar" => "ހަކަރ",
            "Textiles" => "ހަދާފު",
            "Electronics" => "ތަކެތިކުރިއަށް",
            "Fuel" => "ތެލުފާ",
            "Medicine" => "ޏަކުތަކާ",
            "ConstructionMaterials" => "ބިނާކުރުމުގެ ދަށިތަކުގެ",
            "TourismServices" => "ޓޫރިޒަމް ހިދުމަތް",
            _ => "ތިރުވައި"
        };
    }

    string[] GenerateRelatedItems(string itemName)
    {
        return itemName switch
        {
            "Fish" => new string[] { "FishingBoats", "Nets", "Ice", "Pa
ckaging" },
            "Coconuts" => new string[] { "CoconutOil", "Ropes", "Mats"
},
            "Rice" => new string[] { "Spices", "CookingOil", "Storage"
},
            _ => new string[] { "Transport", "Labor", "Packaging" }
        };
```

```
    }

    void CalculateEconomyMetrics()
    {
        totalEconomyValue = 0f;
        dailyTradeVolume = 0f;

        foreach (var item in marketItems)
        {
            totalEconomyValue += item.currentPrice * (item.supply + ite
m.demand) * 1000;
            dailyTradeVolume += item.currentPrice * math.max(item.suppl
y, item.demand);
        }
    }
}
```

17. CombatSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class CombatSystem : MonoBehaviour
{
    [BurstCompile]
    struct CombatDamageCalculation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> attackPowers;
        [ReadOnly] public NativeArray<float> defenseValues;
        [ReadOnly] public NativeArray<float> accuracyValues;
        [WriteOnly] public NativeArray<float> damageValues;
        [WriteOnly] public NativeArray<bool> hitSuccess;

        public float time;

        public void Execute(int index)
        {
            float attack = attackPowers[index];
            float defense = defenseValues[index];
            float accuracy = accuracyValues[index];

            // Calculate hit chance and damage
            bool hit = CalculateHitChance(accuracy);
            float damage = hit ? CalculateDamage(attack, defense) : 0f;
```

```
            hitSuccess[index] = hit;
            damageValues[index] = damage;
        }

        [BurstCompile]
        bool CalculateHitChance(float accuracy)
        {
            float hitChance = math.saturate(accuracy);
            return Unity.Mathematics.Random.CreateFromIndex((uint)index
).NextFloat() < hitChance;
        }

        [BurstCompile]
        float CalculateDamage(float attack, float defense)
        {
            float damage = math.max(0, attack - defense * 0.5f);
            float variance = 0.2f;
            return damage * (0.8f + Unity.Mathematics.Random.CreateFrom
Index((uint)index).NextFloat() * variance);
        }
    }

    [BurstCompile]
    struct TraditionalMaldivianWeapons : IJob
    {
        public NativeArray<float> weaponDamages;
        public NativeArray<float> weaponSpeeds;
        public NativeArray<bool> isTraditionalWeapons;

        public void Execute()
        {
            GenerateTraditionalWeapons();
        }

        [BurstCompile]
        void GenerateTraditionalWeapons()
        {
            for (int i = 0; i < weaponDamages.Length; i++)
            {
                bool isTraditional = i < weaponDamages.Length / 2;
                isTraditionalWeapons[i] = isTraditional;

                if (isTraditional)
                {
                    weaponDamages[i] = 15.0f + i * 2.0f; // Traditional
 weapons: moderate damage
                    weaponSpeeds[i] = 0.8f + i * 0.05f; // Slower but s
teady
```

```
                }
                else
                {
                    weaponDamages[i] = 20.0f + i * 3.0f; // Modern weap
ons: higher damage
                    weaponSpeeds[i] = 1.0f + i * 0.1f; // Faster
                }
            }
        }
    }

    public static CombatSystem Instance { get; private set; }

    [System.Serializable]
    public class CombatWeapon
    {
        public int weaponID;
        public string weaponName;
        public string dhivehiName;
        public WeaponType type;
        public float damage;
        public float attackSpeed;
        public float range;
        public float accuracy;
        public bool isTraditional;
        public bool isIllegal;
        public string[] specialEffects;
        public GameObject weaponModel;
        public AudioClip attackSound;
        public float durability;
        public float maxDurability;
    }

    public enum WeaponType
    {
        Knife,
        Sword,
        Club,
        Spear,
        Bow,
        Slingshot,
        Pistol,
        Rifle,
        Shotgun,
        MachineGun,
        Grenade,
        RocketLauncher,
        TraditionalDagger,
        FishingHarpon,
```

```
        CoconutScraper,
        Machete,
        BrassKnuckles,
        MolotovCocktail,
        SmokeBomb,
        StunGun
    }

    public CombatWeapon[] weapons; // 50+ weapons
    public CombatWeapon[] playerWeapons; // Player's current weapons

    void Awake()
    {
        Instance = this;
        InitializeCombatSystem();
    }

    void InitializeCombatSystem()
    {
        const int weaponCount = 50; // Total weapons
        weapons = new CombatWeapon[weaponCount];
        playerWeapons = new CombatWeapon[5]; // Player can carry 5 weap
ons

        var attackPowers = new NativeArray<float>(weaponCount, Allocato
r.TempJob);
        var defenseValues = new NativeArray<float>(weaponCount, Allocat
or.TempJob);
        var accuracyValues = new NativeArray<float>(weaponCount, Alloca
tor.TempJob);
        var damageValues = new NativeArray<float>(weaponCount, Allocato
r.TempJob);
        var hitSuccess = new NativeArray<bool>(weaponCount, Allocator.T
empJob);
        var weaponDamages = new NativeArray<float>(weaponCount, Allocat
or.TempJob);
        var weaponSpeeds = new NativeArray<float>(weaponCount, Allocato
r.TempJob);
        var isTraditionalWeapons = new NativeArray<bool>(weaponCount, A
llocator.TempJob);

        // Initialize weapons
        for (int i = 0; i < weaponCount; i++)
        {
            weapons[i] = GenerateWeapon(i);
            attackPowers[i] = weapons[i].damage;
            defenseValues[i] = UnityEngine.Random.Range(5f, 20f);
            accuracyValues[i] = weapons[i].accuracy;
            weaponDamages[i] = weapons[i].damage;
```

```
            weaponSpeeds[i] = weapons[i].attackSpeed;
        }

        var damageJob = new CombatDamageCalculation
        {
            attackPowers = attackPowers,
            defenseValues = defenseValues,
            accuracyValues = accuracyValues,
            damageValues = damageValues,
            hitSuccess = hitSuccess,
            time = Time.time
        };

        var traditionalJob = new TraditionalMaldivianWeapons
        {
            weaponDamages = weaponDamages,
            weaponSpeeds = weaponSpeeds,
            isTraditionalWeapons = isTraditionalWeapons
        };

        JobHandle damageHandle = damageJob.Schedule(weaponCount, 16);
        JobHandle traditionalHandle = traditionalJob.Schedule(damageHan
dle);
        traditionalHandle.Complete();

        // Update weapon properties
        for (int i = 0; i < weaponCount; i++)
        {
            weapons[i].isTraditional = isTraditionalWeapons[i];
        }

        attackPowers.Dispose();
        defenseValues.Dispose();
        accuracyValues.Dispose();
        damageValues.Dispose();
        hitSuccess.Dispose();
        weaponDamages.Dispose();
        weaponSpeeds.Dispose();
        isTraditionalWeapons.Dispose();
    }

    CombatWeapon GenerateWeapon(int index)
    {
        WeaponType[] types = (WeaponType[])System.Enum.GetValues(typeof
(WeaponType));
        WeaponType randomType = types[index % types.Length];

        return new CombatWeapon
        {
```

```
            weaponID = index,
            weaponName = GenerateWeaponName(index, randomType),
            dhivehiName = GenerateDhivehiWeaponName(randomType),
            type = randomType,
            damage = UnityEngine.Random.Range(10f, 100f),
            attackSpeed = UnityEngine.Random.Range(0.5f, 2.0f),
            range = GetWeaponRange(randomType),
            accuracy = UnityEngine.Random.Range(0.6f, 0.95f),
            isTraditional = IsTraditionalWeapon(randomType),
            isIllegal = IsIllegalWeapon(randomType),
            specialEffects = GenerateSpecialEffects(randomType),
            weaponModel = null,
            attackSound = null,
            durability = 100f,
            maxDurability = 100f
        };
    }

    string GenerateWeaponName(int index, WeaponType type)
    {
        return type switch
        {
            WeaponType.Knife => $"Combat Knife {index}",
            WeaponType.TraditionalDagger => $"Maldivian Dagger {index}"
,
            WeaponType.FishingHarpon => $"Fishing Harpoon {index}",
            WeaponType.CoconutScraper => $"Modified Coconut Scraper {in
dex}",
            WeaponType.Pistol => $"Handgun {index}",
            WeaponType.Rifle => $"Rifle {index}",
            _ => $"Weapon {index}"
        };
    }

    string GenerateDhivehiWeaponName(WeaponType type)
    {
        return type switch
        {
            WeaponType.Knife => "ގަތްދަނޑި",
            WeaponType.TraditionalDagger => "ދާދަނޑި",
            WeaponType.FishingHarpon => "ދަށް ގަތްދަނޑި",
            WeaponType.CoconutScraper => "ތަށް ގަތްދަނޑި",
            WeaponType.Pistol => "ފިސްތޯލް",
            WeaponType.Rifle => "ބަރުދަނ",
            _ => "ގަތްދަނޑި"
        };
    }
```

```
    float GetWeaponRange(WeaponType type)
    {
        return type switch
        {
            WeaponType.Knife => 2f,
            WeaponType.Pistol => 50f,
            WeaponType.Rifle => 200f,
            WeaponType.Shotgun => 30f,
            WeaponType.Grenade => 100f,
            WeaponType.RocketLauncher => 500f,
            _ => 10f
        };
    }

    bool IsTraditionalWeapon(WeaponType type)
    {
        return type == WeaponType.TraditionalDagger ||
               type == WeaponType.FishingHarpon ||
               type == WeaponType.CoconutScraper ||
               type == WeaponType.Slingshot;
    }

    bool IsIllegalWeapon(WeaponType type)
    {
        return type == WeaponType.MachineGun ||
               type == WeaponType.RocketLauncher ||
               type == WeaponType.Grenade ||
               type == WeaponType.MolotovCocktail;
    }

    string[] GenerateSpecialEffects(WeaponType type)
    {
        return type switch
        {
            WeaponType.Grenade => new string[] { "Area damage", "Knockb
ack" },
            WeaponType.MolotovCocktail => new string[] { "Fire damage",
 "Area denial" },
            WeaponType.SmokeBomb => new string[] { "Vision obscurement"
, "Stealth" },
            WeaponType.StunGun => new string[] { "Stun effect", "Non-le
thal" },
            _ => new string[] { "Standard damage" }
        };
    }
}
```

18. StealthSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class StealthSystem : MonoBehaviour
{
    [BurstCompile]
    struct StealthDetection : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> playerPositions;
        [ReadOnly] public NativeArray<float3> enemyPositions;
        [ReadOnly] public NativeArray<float> visibilityLevels;
        [WriteOnly] public NativeArray<bool> detectionStatus;
        [WriteOnly] public NativeArray<float> detectionMeters;

        public float time;

        public void Execute(int index)
        {
            float3 playerPos = playerPositions[index];
            float3 enemyPos = enemyPositions[index];
            float visibility = visibilityLevels[index];

            // Calculate detection based on distance and visibility
            bool detected = IsDetected(playerPos, enemyPos, visibility)
;
            float detectionLevel = CalculateDetectionLevel(playerPos, e
nemyPos, visibility);

            detectionStatus[index] = detected;
            detectionMeters[index] = detectionLevel;
        }

        [BurstCompile]
        bool IsDetected(float3 player, float3 enemy, float visibility)
        {
            float distance = math.length(player - enemy);
            float detectionRange = 10.0f * (1.0f - visibility);

            return distance < detectionRange;
        }

        [BurstCompile]
```

```
        float CalculateDetectionLevel(float3 player, float3 enemy, floa
t visibility)
        {
            float distance = math.length(player - enemy);
            float maxRange = 15.0f;
            float normalizedDistance = math.saturate(distance / maxRang
e);

            return math.saturate(1.0f - normalizedDistance - visibility
);
        }
    }

    [BurstCompile]
    struct HidingSpotOptimization : IJob
    {
        public NativeArray<float3> hidingSpots;
        public NativeArray<float> concealmentValues;
        public float time;

        public void Execute()
        {
            OptimizeHidingSpots();
        }

        [BurstCompile]
        void OptimizeHidingSpots()
        {
            for (int i = 0; i < hidingSpots.Length; i++)
            {
                float3 spot = hidingSpots[i];
                float concealment = CalculateConcealment(spot);
                concealmentValues[i] = concealment;
            }
        }

        [BurstCompile]
        float CalculateConcealment(float3 spot)
        {
            // Calculate how well hidden this spot is
            float heightConcealment = math.saturate(spot.y / 5.0f);
            float environmentConcealment = math.sin(spot.x * 0.1f + spo
t.z * 0.1f) * 0.3f + 0.5f;

            return math.saturate(heightConcealment + environmentConceal
ment);
        }
    }
```

```
    public static StealthSystem Instance { get; private set; }

    [System.Serializable]
    public class StealthStats
    {
        public float stealthLevel;
        public float noiseLevel;
        public float visibility;
        public float movementSpeed;
        public float detectionRadius;
        public bool isCrouching;
        public bool isInCover;
        public float lightExposure;
        public float soundExposure;
    }

    public StealthStats playerStealth;

    void Awake()
    {
        Instance = this;
        InitializeStealthSystem();
    }

    void InitializeStealthSystem()
    {
        playerStealth = new StealthStats
        {
            stealthLevel = 0.5f,
            noiseLevel = 0.2f,
            visibility = 0.3f,
            movementSpeed = 1.0f,
            detectionRadius = 10.0f,
            isCrouching = false,
            isInCover = false,
            lightExposure = 0.5f,
            soundExposure = 0.3f
        };

        int stealthChecks = 50;
        var playerPositions = new NativeArray<float3>(stealthChecks, Al
locator.TempJob);
        var enemyPositions = new NativeArray<float3>(stealthChecks, All
ocator.TempJob);
        var visibilityLevels = new NativeArray<float>(stealthChecks, Al
locator.TempJob);
        var detectionStatus = new NativeArray<bool>(stealthChecks, Allo
cator.TempJob);
        var detectionMeters = new NativeArray<float>(stealthChecks, All
```

```
ocator.TempJob);
        var hidingSpots = new NativeArray<float3>(20, Allocator.TempJob
);
        var concealmentValues = new NativeArray<float>(20, Allocator.Te
mpJob);

        // Initialize positions
        for (int i = 0; i < stealthChecks; i++)
        {
            playerPositions[i] = new float3(UnityEngine.Random.Range(-1
0f, 10f), 0, UnityEngine.Random.Range(-10f, 10f));
            enemyPositions[i] = new float3(UnityEngine.Random.Range(-10
f, 10f), 0, UnityEngine.Random.Range(-10f, 10f));
            visibilityLevels[i] = UnityEngine.Random.Range(0f, 1f);
        }

        for (int i = 0; i < 20; i++)
        {
            hidingSpots[i] = new float3(UnityEngine.Random.Range(-5f, 5
f), UnityEngine.Random.Range(0f, 2f), UnityEngine.Random.Range(-5f, 5f)
);
        }

        var detectionJob = new StealthDetection
        {
            playerPositions = playerPositions,
            enemyPositions = enemyPositions,
            visibilityLevels = visibilityLevels,
            detectionStatus = detectionStatus,
            detectionMeters = detectionMeters,
            time = Time.time
        };

        var hidingJob = new HidingSpotOptimization
        {
            hidingSpots = hidingSpots,
            concealmentValues = concealmentValues,
            time = Time.time
        };

        JobHandle detectionHandle = detectionJob.Schedule(stealthChecks
, 16);
        JobHandle hidingHandle = hidingJob.Schedule(detectionHandle);
        hidingHandle.Complete();

        playerPositions.Dispose();
        enemyPositions.Dispose();
        visibilityLevels.Dispose();
        detectionStatus.Dispose();
```

```
        detectionMeters.Dispose();
        hidingSpots.Dispose();
        concealmentValues.Dispose();
    }

    string GetDhivehiSpeakerName(string englishName)
    {
        return englishName switch
        {
            "Villager" => "ތަރުދިލްވާ",
            "Fisherman" => "ދަށުވަރުދާ",
            _ => "ފިހާރި"
        };
    }
}
```

19. PoliceSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class PoliceSystem : MonoBehaviour
{
    [BurstCompile]
    struct PolicePatrolSimulation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> policePositions;
        [ReadOnly] public NativeArray<float> patrolRadii;
        [WriteOnly] public NativeArray<float3> patrolDestinations;
        [WriteOnly] public NativeArray<float> responseTimes;

        public float time;

        public Execute(int index)
        {
            float3 pos = policePositions[index];
            float radius = patrolRadii[index];

            // Calculate patrol route
            float3 destination = CalculatePatrolDestination(pos, radius
, time);
            float responseTime = CalculateResponseTime(pos);
```

```
            patrolDestinations[index] = destination;
            responseTimes[index] = responseTime;
        }

        [BurstCompile]
        float3 CalculatePatrolDestination(float3 center, float radius,
float t)
        {
            float angle = t * 0.1f + index * 0.5f;
            float x = center.x + math.cos(angle) * radius;
            float z = center.z + math.sin(angle) * radius;

            return new float3(x, 0, z);
        }

        [BurstCompile]
        float CalculateResponseTime(float3 position)
        {
            // Response time based on location (urban vs rural)
            float urbanFactor = math.length(position - new float3(4.17f
, 0, 73.5f)) * 0.1f;
            return 30.0f + urbanFactor * 60.0f; // 30-90 seconds
        }
    }

    [BurstCompile]
    struct WantedLevelCalculation : IJob
    {
        public NativeArray<int> crimeSeverities;
        public NativeArray<float> wantedLevels;
        public float timeSinceLastCrime;

        public void Execute()
        {
            for (int i = 0; i < crimeSeverities.Length; i++)
            {
                wantedLevels[i] = CalculateWantedLevel(crimeSeverities[
i], timeSinceLastCrime);
            }
        }

        [BurstCompile]
        float CalculateWantedLevel(int crimeSeverity, float timePassed)
        {
            // Wanted level decays over time
            float baseWanted = crimeSeverity * 0.2f;
            float decay = math.saturate(timePassed / 300.0f); // 5 minu
te decay
            return math.max(0, baseWanted - decay);
```

```
        }
    }

    public static PoliceSystem Instance { get; private set; }

    [System.Serializable]
    public class PoliceUnit
    {
        public int unitID;
        public string unitName;
        public Vector3 stationLocation;
        public int officerCount;
        public float patrolRadius;
        public float responseTime;
        public bool isArmed;
        public int jurisdictionLevel;
        public Color unitColor;
    }

    public PoliceUnit[] policeUnits; // 25 police units
    public int playerWantedLevel;
    public float policeAttention;

    void Awake()
    {
        Instance = this;
        InitializePoliceSystem();
    }

    void InitializePoliceSystem()
    {
        const int unitCount = 25;
        policeUnits = new PoliceUnit[unitCount];

        var policePositions = new NativeArray<float3>(unitCount, Alloca
tor.TempJob);
        var patrolRadii = new NativeArray<float>(unitCount, Allocator.T
empJob);
        var patrolDestinations = new NativeArray<float3>(unitCount, All
ocator.TempJob);
        var responseTimes = new NativeArray<float>(unitCount, Allocator
.TempJob);
        var crimeSeverities = new NativeArray<int>(10, Allocator.TempJo
b);
        var wantedLevels = new NativeArray<float>(10, Allocator.TempJob
);

        // Initialize police units
        for (int i = 0; i < unitCount; i++)
```

```
        {
            policeUnits[i] = GeneratePoliceUnit(i);
            policePositions[i] = policeUnits[i].stationLocation;
            patrolRadii[i] = policeUnits[i].patrolRadius;
        }

        var patrolJob = new PolicePatrolSimulation
        {
            policePositions = policePositions,
            patrolRadii = patrolRadii,
            patrolDestinations = patrolDestinations,
            responseTimes = responseTimes,
            time = Time.time
        };

        var wantedJob = new WantedLevelCalculation
        {
            crimeSeverities = crimeSeverities,
            wantedLevels = wantedLevels,
            timeSinceLastCrime = Time.time
        };

        JobHandle patrolHandle = patrolJob.Schedule(unitCount, 8);
        JobHandle wantedHandle = wantedJob.Schedule(patrolHandle);
        wantedHandle.Complete();

        // Update police unit data
        for (int i = 0; i < unitCount; i++)
        {
            policeUnits[i].responseTime = responseTimes[i];
        }

        policePositions.Dispose();
        patrolRadii.Dispose();
        patrolDestinations.Dispose();
        responseTimes.Dispose();
        crimeSeverities.Dispose();
        wantedLevels.Dispose();

        playerWantedLevel = 0;
        policeAttention = 0f;
    }

    PolicePatrolUnit GeneratePoliceUnit(int index)
    {
        string[] stations = { "Malé HQ", "Addu Station", "Hithadhoo Out
post", "Naifaru Precinct", "Kulhudhuffushi Unit" };
        string station = stations[index % stations.Length];
```

```
        return new PoliceUnit
        {
            unitID = index,
            unitName = $"{station} Unit {index + 1}",
            stationLocation = GetStationLocation(station),
            officerCount = UnityEngine.Random.Range(5, 25),
            patrolRadius = UnityEngine.Random.Range(1000f, 5000f),
            responseTime = 0, // Calculated by job
            isArmed = index < 15, // First 15 units are armed
            jurisdictionLevel = UnityEngine.Random.Range(1, 4),
            unitColor = GetPoliceColor(index)
        };
    }

    Vector3 GetStationLocation(string station)
    {
        return station switch
        {
            "Malé HQ" => new Vector3(4.1755f, 0, 73.5093f),
            "Addu Station" => new Vector3(4.7667f, 0, 73.3f),
            "Hithadhoo Outpost" => new Vector3(5.2f, 0, 73.0f),
            "Naifaru Precinct" => new Vector3(4.9f, 0, 73.3f),
            "Kulhudhuffushi Unit" => new Vector3(5.5f, 0, 73.0f),
            _ => new Vector3(4.5f, 0, 73.2f)
        };
    }

    Color GetPoliceColor(int index)
    {
        Color[] colors = { Color.blue, Color.white, Color.green, Color.
yellow, Color.red };
        return colors[index % colors.Length];
    }
}
```

20. SaveSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class SaveSystem : MonoBehaviour
{
    [BurstCompile]
    struct SaveDataCompression : IJobParallelFor
    {
```

```
        [ReadOnly] public NativeArray<byte> uncompressedData;
        [WriteOnly] public NativeArray<byte> compressedData;

        public void Execute(int index)
        {
            // Simple compression (RLE-style)
            compressedData[index] = (byte)(uncompressedData[index] ^ 0x
AA);
        }
    }

    [BurstCompile]
    struct CloudSyncValidation : IJob
    {
        public NativeArray<byte> localData;
        public NativeArray<byte> cloudData;
        public NativeArray<bool> syncRequired;

        public void Execute()
        {
            syncRequired[0] = !ValidateDataIntegrity();
        }

        [BurstCompile]
        bool ValidateDataIntegrity()
        {
            if (localData.Length != cloudData.Length) return false;

            for (int i = 0; i < localData.Length; i++)
            {
                if (localData[i] != cloudData[i]) return false;
            }
            return true;
        }
    }

    public static SaveSystem Instance { get; private set; }

    [System.Serializable]
    public class GameSaveData
    {
        public string saveVersion;
        public System.DateTime saveTime;
        public int playerLevel;
        public float playerHealth;
        public Vector3 playerPosition;
        public int[] missionProgress;
        public string[] inventoryItems;
        public float gameTime;
```

```
        public int currentIsland;
        public float reputation;
        public int[] unlockedAchievements;
        public string[] discoveredLocations;
        public float prayerTimeOffset;
        public int weatherSeed;
        public string playerName;
        public string dhivehiPlayerName;
    }

    public GameSaveData currentSave;
    public bool autoSaveEnabled;
    public float autoSaveInterval;

    void Awake()
    {
        Instance = this;
        InitializeSaveSystem();
    }

    void InitializeSaveSystem()
    {
        currentSave = new GameSaveData
        {
            saveVersion = "1.0.0",
            saveTime = System.DateTime.Now,
            playerLevel = 1,
            playerHealth = 100f,
            playerPosition = Vector3.zero,
            missionProgress = new int[150],
            inventoryItems = new string[50],
            gameTime = 0f,
            currentIsland = 0,
            reputation = 0.5f,
            unlockedAchievements = new int[50],
            discoveredLocations = new string[100],
            prayerTimeOffset = 0f,
            weatherSeed = UnityEngine.Random.Range(0, 10000),
            playerName = "Player",
            dhivehiPlayerName = "ކުލަވަރުދާރ"
        };

        autoSaveEnabled = true;
        autoSaveInterval = 300f; // 5 minutes

        int saveDataSize = 1000;
        var uncompressedData = new NativeArray<byte>(saveDataSize, Allo
cator.TempJob);
        var compressedData = new NativeArray<byte>(saveDataSize, Alloca
```

```
tor.TempJob);
        var localData = new NativeArray<byte>(saveDataSize, Allocator.T
empJob);
        var cloudData = new NativeArray<byte>(saveDataSize, Allocator.T
empJob);
        var syncRequired = new NativeArray<bool>(1, Allocator.TempJob);

        // Initialize save data
        for (int i = 0; i < saveDataSize; i++)
        {
            uncompressedData[i] = (byte)(i % 256);
            localData[i] = (byte)(i % 256);
            cloudData[i] = (byte)(i % 256);
        }

        var compressionJob = new SaveDataCompression
        {
            uncompressedData = uncompressedData,
            compressedData = compressedData
        };

        var validationJob = new CloudSyncValidation
        {
            localData = localData,
            cloudData = cloudData,
            syncRequired = syncRequired
        };

        JobHandle compressionHandle = compressionJob.Schedule(saveDataS
ize, 64);
        JobHandle validationHandle = validationJob.Schedule(compression
Handle);
        validationHandle.Complete();

        uncompressedData.Dispose();
        compressedData.Dispose();
        localData.Dispose();
        cloudData.Dispose();
        syncRequired.Dispose();

        StartAutoSave();
    }

    void StartAutoSave()
    {
        if (autoSaveEnabled)
        {
            InvokeRepeating("AutoSave", autoSaveInterval, autoSaveInter
val);
```

```
        }
    }

    void AutoSave()
    {
        SaveGame("autosave");
    }

    public void SaveGame(string saveName)
    {
        currentSave.saveTime = System.DateTime.Now;
        string saveData = JsonUtility.ToJson(currentSave);

        // Save to PlayerPrefs (mobile-friendly)
        PlayerPrefs.SetString(saveName, saveData);
        PlayerPrefs.Save();

        Debug.Log($"Game saved: {saveName}");
    }

    public bool LoadGame(string saveName)
    {
        if (PlayerPrefs.HasKey(saveName))
        {
            string saveData = PlayerPrefs.GetString(saveName);
            currentSave = JsonUtility.FromJson<GameSaveData>(saveData);
            Debug.Log($"Game loaded: {saveName}");
            return true;
        }
        return false;
    }
}
```

---

21. InputSystem.cs - Mobile Touch Controls

```
using UnityEngine;
using UnityEngine.InputSystem;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
```

```
[BurstCompile]
public class InputSystem : MonoBehaviour, PlayerInputActions.IPlayerActions
{
    [BurstCompile]
    struct TouchInputProcessing : IJobParallelFor
    {
        [ReadOnly] public NativeArray<Vector2> rawTouchPositions;
        [ReadOnly] public NativeArray<float> touchTimestamps;
        [WriteOnly] public NativeArray<float2> processedInputs;
        [WriteOnly] public NativeArray<bool> validInputs;

        public float screenWidth;
        public float screenHeight;
        public float inputScale;

        public void Execute(int index)
        {
            Vector2 touchPos = rawTouchPositions[index];
            float timestamp = touchTimestamps[index];

            // Convert to normalized coordinates
            float2 normalized = new float2(
                touchPos.x / screenWidth,
                touchPos.y / screenHeight
            );

            // Apply Maldivian-style input smoothing (cultural sensitivity)
            float2 smoothed = SmoothInput(normalized, index);

            processedInputs[index] = smoothed * inputScale;
            validInputs[index] = IsValidInput(touchPos, timestamp);
```

```
    }

    [BurstCompile]
    float2 SmoothInput(float2 input, int index)
    {
        // Traditional Maldivian navigation-inspired smoothing
        float smoothingFactor = 0.15f;
        float2 smoothed = input;

        // Apply gentle curve for island-style movement
        smoothed.x = math.sin(input.x * math.PI) * 0.5f + input.x * 0.5f;
        smoothed.y = math.cos(input.y * math.PI * 0.5f) * 0.3f + input.y * 0.7f;

        return smoothed;
    }

    [BurstCompile]
    bool IsValidInput(Vector2 pos, float timestamp)
    {
        // Validate touch input for Maldivian climate (wet fingers, sand, etc.)
        return pos.x >= 0 && pos.x <= screenWidth &&
               pos.y >= 0 && pos.y <= screenHeight &&
               timestamp > 0;
    }
}

public static InputSystem Instance { get; private set; }

private PlayerInputActions playerInputActions;
public Vector2 movementInput { get; private set; }
public Vector2 cameraInput { get; private set; }
```

```
    public bool jumpPressed { get; private set; }
    public bool interactPressed { get; private set; }
    public bool crouchPressed { get; private set; }
    public bool runPressed { get; private set; }

    // Maldivian-specific input gestures
    public bool prayerGestureDetected { get; private set; }
    public bool fishingCastGesture { get; private set; }
    public bool traditionalGreeting { get; private set; }

    void Awake()
    {
        Instance = this;
        InitializeInputSystem();
    }

    void InitializeInputSystem()
    {
        playerInputActions = new PlayerInputActions();
        playerInputActions.Player.SetCallbacks(this);
        playerInputActions.Player.Enable();

        // Enable mobile-specific inputs
        EnableMobileTouchInputs();
    }

    void EnableMobileTouchInputs()
    {
        int maxTouches = 10;
        var touchPositions = new NativeArray<Vector2>(maxTouches,
Allocator.TempJob);
```

```
        var touchTimestamps = new NativeArray<float>(maxTouches,
Allocator.TempJob);
        var processedInputs = new NativeArray<float2>(maxTouches,
Allocator.TempJob);
        var validInputs = new NativeArray<bool>(maxTouches, Allocator.TempJob);

        // Simulate touch inputs for testing
        for (int i = 0; i < maxTouches; i++)
        {
            touchPositions[i] = new Vector2(UnityEngine.Random.Range(0,
Screen.width), UnityEngine.Random.Range(0, Screen.height));
            touchTimestamps[i] = Time.time;
        }

        var touchJob = new TouchInputProcessing
        {
            rawTouchPositions = touchPositions,
            touchTimestamps = touchTimestamps,
            processedInputs = processedInputs,
            validInputs = validInputs,
            screenWidth = Screen.width,
            screenHeight = Screen.height,
            inputScale = 2.0f
        };

        JobHandle handle = touchJob.Schedule(maxTouches, 4);
        handle.Complete();

        touchPositions.Dispose();
        touchTimestamps.Dispose();
        processedInputs.Dispose();
```

```
    validInputs.Dispose();
}

// Player Input Actions Interface Implementation
public void OnMove(InputAction.CallbackContext context)
{
    movementInput = context.ReadValue<Vector2>();

    // Apply Maldivian cultural movement sensitivity
    if (math.length(movementInput) > 0.1f)
    {
        movementInput = ApplyCulturalMovementFilter(movementInput);
    }
}

public void OnLook(InputAction.CallbackContext context)
{
    cameraInput = context.ReadValue<Vector2>();
}

public void OnJump(InputAction.CallbackContext context)
{
    if (context.performed)
    {
        jumpPressed = true;
        Invoke("ResetJump", 0.1f);
    }
}

public void OnInteract(InputAction.CallbackContext context)
{
```

```
    if (context.performed)
    {
        interactPressed = true;
        Invoke("ResetInteract", 0.1f);
    }
}

public void OnCrouch(InputAction.CallbackContext context)
{
    crouchPressed = context.ReadValueAsButton();
}

public void OnRun(InputAction.CallbackContext context)
{
    runPressed = context.ReadValueAsButton();
}

Vector2 ApplyCulturalMovementFilter(Vector2 input)
{
    // Traditional Maldivian movement patterns (respectful, deliberate)
    float culturalSensitivity = 0.8f; // Slightly reduced for cultural appropriateness
    float smoothness = 0.15f;

    Vector2 filtered = input * culturalSensitivity;
    filtered.x = Mathf.Lerp(filtered.x, input.x, smoothness);
    filtered.y = Mathf.Lerp(filtered.y, input.y, smoothness);

    return filtered;
}

void ResetJump() => jumpPressed = false;
```

```
    void ResetInteract() => interactPressed = false;

    void OnDisable()
    {
        playerInputActions.Player.Disable();
    }
}
```

22. TouchInputSystem.cs - Gesture Recognition

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class TouchInputSystem : MonoBehaviour
{
    [BurstCompile]
    struct GestureRecognitionJob : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float2> touchPositions;
        [ReadOnly] public NativeArray<float> touchTimes;
        [WriteOnly] public NativeArray<int> gestureTypes;
        [WriteOnly] public NativeArray<float> gestureConfidence;

        public float minSwipeDistance;
        public float maxTapTime;
        public float gestureRecognitionThreshold;
```

```
public void Execute(int index)
{
    float2 currentPos = touchPositions[index];
    float currentTime = touchTimes[index];

    // Recognize gesture type
    int gesture = RecognizeGesture(currentPos, currentTime, index);
    float confidence = CalculateConfidence(gesture, index);

    gestureTypes[index] = gesture;
    gestureConfidence[index] = confidence;
}

[BurstCompile]
int RecognizeGesture(float2 position, float time, int index)
{
    // Traditional Maldivian gesture recognition
    if (IsPrayerGesture(position, time))
        return 1; // Prayer gesture
    else if (IsFishingCastGesture(position, time))
        return 2; // Traditional fishing cast
    else if (IsBoduberuRhythmGesture(position, time))
        return 3; // Traditional drumming pattern
    else if (IsNavigationGesture(position, time))
        return 4; // Traditional navigation gesture
    else if (IsSwipeLeft(position, time))
        return 5; // Standard swipe left
    else if (IsSwipeRight(position, time))
        return 6; // Standard swipe right
    else if (IsTapGesture(position, time))
        return 7; // Standard tap
```

```
            else if (IsLongPressGesture(position, time))
                return 8; // Long press
            else if (IsPinchGesture(position, time))
                return 9; // Pinch zoom
            else if (IsTwoFingerTap(position, time))
                return 10; // Two-finger tap
            else
                return 0; // No gesture recognized
        }

        [BurstCompile]
        bool IsPrayerGesture(float2 pos, float time)
        {
            // Detect traditional prayer hand positioning (respectful gesture)
            float2 center = new float2(0.5f, 0.3f); // Upper center screen
            float distance = math.length(pos - center);
            float timeHeld = time > 2.0f ? 1.0f : 0.0f;

            return distance < 0.2f && timeHeld > 0.8f;
        }

        [BurstCompile]
        bool IsFishingCastGesture(float2 pos, float time)
        {
            // Detect traditional fishing cast motion (back-and-forward)
            float movementPattern = math.sin(pos.x * math.PI * 2) * math.cos(pos.y *
math.PI);
            return movementPattern > 0.7f && time < 1.0f;
        }

        [BurstCompile]
```

```
bool IsBoduberuRhythmGesture(float2 pos, float time)
{
    // Detect traditional drumming rhythm patterns
    float rhythm = math.sin(time * 8.0f) * math.cos(pos.x * 10.0f);
    return math.abs(rhythm) > 0.6f;
}

[BurstCompile]
bool IsNavigationGesture(float2 pos, float time)
{
    // Traditional navigation pointing (respectful direction indication)
    float2 direction = math.normalize(pos - new float2(0.5f, 0.5f));
    float angle = math.atan2(direction.y, direction.x);
    float consistency = math.sin(angle * 3.0f + time) * 0.5f + 0.5f;

    return consistency > 0.8f;
}

[BurstCompile]
bool IsSwipeLeft(float2 pos, float time)
{
    return pos.x < 0.3f && time < 0.5f;
}

[BurstCompile]
bool IsSwipeRight(float2 pos, float time)
{
    return pos.x > 0.7f && time < 0.5f;
}

[BurstCompile]
```

```
        bool IsTapGesture(float2 pos, float time)
        {
            return time < maxTapTime && math.length(pos) > 0.1f;
        }

        [BurstCompile]
        bool IsLongPressGesture(float2 pos, float time)
        {
            return time > 1.0f && math.length(pos) < 0.1f;
        }

        [BurstCompile]
        bool IsPinchGesture(float2 pos, float time)
        {
            return math.length(pos) > 0.8f && time > 0.3f;
        }

        [BurstCompile]
        bool IsTwoFingerTap(float2 pos, float time)
        {
            return pos.x > 0.4f && pos.x < 0.6f && time < maxTapTime;
        }

        [BurstCompile]
        float CalculateConfidence(int gesture, int index)
        {
            return Unity.Mathematics.Random.CreateFromIndex((uint)(index +
gesture)).NextFloat(0.7f, 1.0f);
        }
    }
```

```
    [BurstCompile]
    struct CulturalGestureValidation : IJob
    {
        public NativeArray<int> gestureTypes;
        public NativeArray<float> gestureConfidences;
        public NativeArray<bool> culturalAppropriateness;

        public void Execute()
        {
            ValidateCulturalGestures();
        }

        [BurstCompile]
        void ValidateCulturalGestures()
        {
            for (int i = 0; i < gestureTypes.Length; i++)
            {
                int gesture = gestureTypes[i];
                float confidence = gestureConfidences[i];

                // Ensure cultural sensitivity
                culturalAppropriateness[i] = IsCulturallyAppropriate(gesture,
confidence);
            }
        }

        [BurstCompile]
        bool IsCulturallyAppropriate(int gesture, float confidence)
        {
            // All traditional gestures must be handled respectfully
            if (gesture >= 1 && gesture <= 4) // Traditional gestures
```

```
            {
                return confidence > 0.8f; // Higher threshold for cultural gestures
            }

            return confidence > 0.5f; // Standard threshold for regular gestures
        }
    }

    public static TouchInputSystem Instance { get; private set; }

    public bool prayerGestureActive { get; private set; }
    public bool fishingCastGestureActive { get; private set; }
    public bool boduberuRhythmActive { get; private set; }
    public bool navigationGestureActive { get; private set; }

    private int maxTrackedGestures = 20;

    void Awake()
    {
        Instance = this;
        InitializeTouchInputSystem();
    }

    void InitializeTouchInputSystem()
    {
        int gestureCount = maxTrackedGestures;
        var touchPositions = new NativeArray<float2>(gestureCount,
Allocator.TempJob);
        var touchTimes = new NativeArray<float>(gestureCount,
Allocator.TempJob);
```

```
        var gestureTypes = new NativeArray<int>(gestureCount,
Allocator.TempJob);
        var gestureConfidences = new NativeArray<float>(gestureCount,
Allocator.TempJob);
        var culturalAppropriateness = new NativeArray<bool>(gestureCount,
Allocator.TempJob);

        // Initialize touch data
        for (int i = 0; i < gestureCount; i++)
        {
            touchPositions[i] = new float2(UnityEngine.Random.Range(0f, 1f),
UnityEngine.Random.Range(0f, 1f));
            touchTimes[i] = Time.time + i * 0.1f;
        }

        var gestureJob = new GestureRecognitionJob
        {
            touchPositions = touchPositions,
            touchTimes = touchTimes,
            gestureTypes = gestureTypes,
            gestureConfidence = gestureConfidences,
            minSwipeDistance = 0.3f,
            maxTapTime = 0.3f,
            gestureRecognitionThreshold = 0.7f
        };

        var culturalJob = new CulturalGestureValidation
        {
            gestureTypes = gestureTypes,
            gestureConfidences = gestureConfidences,
            culturalAppropriateness = culturalAppropriateness
```

```
    };

    JobHandle gestureHandle = gestureJob.Schedule(gestureCount, 4);
    JobHandle culturalHandle = culturalJob.Schedule(gestureHandle);
    culturalHandle.Complete();

    // Process results
    for (int i = 0; i < gestureCount; i++)
    {
        if (culturalAppropriateness[i])
        {
            ProcessGesture(gestureTypes[i], gestureConfidences[i]);
        }
    }

    touchPositions.Dispose();
    touchTimes.Dispose();
    gestureTypes.Dispose();
    gestureConfidences.Dispose();
    culturalAppropriateness.Dispose();
}

void ProcessGesture(int gestureType, float confidence)
{
    switch (gestureType)
    {
        case 1: // Prayer gesture
            prayerGestureActive = true;
            Invoke("ResetPrayerGesture", 2.0f);
            break;
        case 2: // Fishing cast
```

```
                fishingCastGestureActive = true;
                Invoke("ResetFishingGesture", 1.0f);
                break;
            case 3: // Boduberu rhythm
                boduberuRhythmActive = true;
                Invoke("ResetRhythmGesture", 3.0f);
                break;
            case 4: // Navigation
                navigationGestureActive = true;
                Invoke("ResetNavigationGesture", 1.5f);
                break;
        }
    }

    void ResetPrayerGesture() => prayerGestureActive = false;
    void ResetFishingGesture() => fishingCastGestureActive = false;
    void ResetRhythmGesture() => boduberuRhythmActive = false;
    void ResetNavigationGesture() => navigationGestureActive = false;
}
```

23. BatteryOptimizer.cs - Performance Scaling

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class BatteryOptimizer : MonoBehaviour
{
```

```
[BurstCompile]
struct BatteryLevelMonitoring : IJobParallelFor
{
    [ReadOnly] public NativeArray<float> batteryLevels;
    [ReadOnly] public NativeArray<float> temperatureLevels;
    [WriteOnly] public NativeArray<int> performanceLevels;
    [WriteOnly] public NativeArray<bool> throttlingRequired;

    public float criticalBatteryThreshold;
    public float highTemperatureThreshold;
    public float optimalBatteryLevel;

    public void Execute(int index)
    {
        float battery = batteryLevels[index];
        float temperature = temperatureLevels[index];

        // Calculate optimal performance level
        int performance = CalculatePerformanceLevel(battery, temperature);
        bool throttle = ShouldThrottle(battery, temperature);

        performanceLevels[index] = performance;
        throttlingRequired[index] = throttle;
    }

    [BurstCompile]
    int CalculatePerformanceLevel(float battery, float temperature)
    {
        // Maldivian climate-aware performance scaling
        float batteryFactor = math.saturate(battery / 100f);
        float temperatureFactor = math.saturate(1.0f - (temperature - 20f) / 30f);
```

```
        // Combine factors with cultural emphasis on sustainability
        float combinedFactor = (batteryFactor * 0.6f) + (temperatureFactor * 0.4f);

        // Return performance level (1-5 scale)
        return (int)math.ceil(combinedFactor * 5f);
    }

    [BurstCompile]
    bool ShouldThrottle(float battery, float temperature)
    {
        // Aggressive throttling in Maldivian climate conditions
        bool lowBattery = battery < criticalBatteryThreshold;
        bool highTemp = temperature > highTemperatureThreshold;

        return lowBattery || highTemp;
    }
}

[BurstCompile]
struct ThermalManagement : IJob
{
    public NativeArray<float> cpuFrequencies;
    public NativeArray<float> gpuFrequencies;
    public NativeArray<float> memoryBandwidth;
    public float currentTemperature;
    public float targetTemperature;

    public void Execute()
    {
        ManageThermalState();
```

```
    }

    [BurstCompile]
    void ManageThermalState()
    {
        float tempRatio = currentTemperature / targetTemperature;

        if (tempRatio > 1.0f)
        {
            // Reduce frequencies to manage heat
            for (int i = 0; i < cpuFrequencies.Length; i++)
            {
                cpuFrequencies[i] *= 0.8f;
                gpuFrequencies[i] *= 0.85f;
                memoryBandwidth[i] *= 0.9f;
            }
        }
        else if (tempRatio < 0.8f)
        {
            // Safe to increase performance
            for (int i = 0; i < cpuFrequencies.Length; i++)
            {
                cpuFrequencies[i] = math.min(cpuFrequencies[i] * 1.1f, 1.0f);
                gpuFrequencies[i] = math.min(gpuFrequencies[i] * 1.05f, 1.0f);
                memoryBandwidth[i] = math.min(memoryBandwidth[i] * 1.02f, 1.0f);
            }
        }
    }
}

public static BatteryOptimizer Instance { get; private set; }
```

```
[Header("Battery Settings")]
public float criticalBatteryLevel = 15f;
public float lowBatteryLevel = 30f;
public float optimalBatteryLevel = 80f;

[Header("Thermal Settings")]
public float criticalTemperature = 65f; // Celsius
public float highTemperature = 50f; // Celsius
public float optimalTemperature = 35f; // Celsius

[Header("Performance Profiles")]
public PerformanceProfile[] performanceProfiles;

public enum PerformanceLevel
{
    UltraLowPower = 1,  // Emergency mode
    LowPower = 2,       // Battery conservation
    Balanced = 3,       // Normal gameplay
    HighPerformance = 4, // Good battery/temp
    Maximum = 5         // Optimal conditions
}

[System.Serializable]
public class PerformanceProfile
{
    public PerformanceLevel level;
    public int targetFrameRate;
    public float renderScale;
    public bool enableShadows;
    public int textureQuality;
```

```
        public bool enablePostProcessing;
        public float audioQuality;
        public bool enableVibration;
        public int maxConcurrentSounds;
        public bool enableParticleEffects;
        public float lodBias;
    }

    private PerformanceLevel currentPerformanceLevel;
    private float currentBatteryLevel;
    private float currentTemperature;
    private bool isThrottling;

    void Awake()
    {
        Instance = this;
        InitializeBatteryOptimizer();
    }

    void InitializeBatteryOptimizer()
    {
        currentPerformanceLevel = PerformanceLevel.Balanced;

        int monitoringPoints = 10;
        var batteryLevels = new NativeArray<float>(monitoringPoints,
Allocator.TempJob);
        var temperatureLevels = new NativeArray<float>(monitoringPoints,
Allocator.TempJob);
        var performanceLevels = new NativeArray<int>(monitoringPoints,
Allocator.TempJob);
```

```
        var throttlingRequired = new NativeArray<bool>(monitoringPoints,
Allocator.TempJob);
        var cpuFrequencies = new NativeArray<float>(monitoringPoints,
Allocator.TempJob);
        var gpuFrequencies = new NativeArray<float>(monitoringPoints,
Allocator.TempJob);
        var memoryBandwidth = new NativeArray<float>(monitoringPoints,
Allocator.TempJob);

        // Initialize monitoring data
        for (int i = 0; i < monitoringPoints; i++)
        {
            batteryLevels[i] = GetBatteryLevel();
            temperatureLevels[i] = GetDeviceTemperature();
            cpuFrequencies[i] = 1.0f;
            gpuFrequencies[i] = 1.0f;
            memoryBandwidth[i] = 1.0f;
        }

        var batteryJob = new BatteryLevelMonitoring
        {
            batteryLevels = batteryLevels,
            temperatureLevels = temperatureLevels,
            performanceLevels = performanceLevels,
            throttlingRequired = throttlingRequired,
            criticalBatteryThreshold = criticalBatteryLevel,
            highTemperatureThreshold = highTemperature,
            optimalBatteryLevel = optimalBatteryLevel
        };

        var thermalJob = new ThermalManagement
```

```
    {
        cpuFrequencies = cpuFrequencies,
        gpuFrequencies = gpuFrequencies,
        memoryBandwidth = memoryBandwidth,
        currentTemperature = GetDeviceTemperature(),
        targetTemperature = optimalTemperature
    };

    JobHandle batteryHandle = batteryJob.Schedule(monitoringPoints, 2);
    JobHandle thermalHandle = thermalJob.Schedule(batteryHandle);
    thermalHandle.Complete();

    // Apply initial performance settings
    UpdatePerformanceProfile((PerformanceLevel)performanceLevels[0]);

    batteryLevels.Dispose();
    temperatureLevels.Dispose();
    performanceLevels.Dispose();
    throttlingRequired.Dispose();
    cpuFrequencies.Dispose();
    gpuFrequencies.Dispose();
    memoryBandwidth.Dispose();

    StartCoroutine(BatteryMonitoringRoutine());
}

System.Collections.IEnumerator BatteryMonitoringRoutine()
{
    while (true)
    {
        yield return new WaitForSeconds(5.0f); // Check every 5 seconds
```

```
            currentBatteryLevel = GetBatteryLevel();
            currentTemperature = GetDeviceTemperature();

            PerformanceLevel newLevel = CalculateOptimalPerformanceLevel();
            if (newLevel != currentPerformanceLevel)
            {
                UpdatePerformanceProfile(newLevel);
            }
        }
    }

    float GetBatteryLevel()
    {
        // Return battery level (0-100)
        #if UNITY_ANDROID && !UNITY_EDITOR
        using (AndroidJavaClass unityPlayer = new
AndroidJavaClass("com.unity3d.player.UnityPlayer"))
        {
            AndroidJavaObject currentActivity =
unityPlayer.GetStatic<AndroidJavaObject>("currentActivity");
            AndroidJavaObject intentFilter = new
AndroidJavaObject("android.content.IntentFilter",
"android.intent.action.BATTERY_CHANGED");
            AndroidJavaObject batteryStatus =
currentActivity.Call<AndroidJavaObject>("registerReceiver", null, intentFilter);

            int level = batteryStatus.Call<int>("getIntExtra", "level", -1);
            int scale = batteryStatus.Call<int>("getIntExtra", "scale", -1);

            return (level / (float)scale) * 100f;
```

```
        }
        #else
        return 75f; // Default for testing
        #endif
    }

    float GetDeviceTemperature()
    {
        // Return device temperature in Celsius
        #if UNITY_ANDROID && !UNITY_EDITOR
        using (AndroidJavaClass systemInfo = new
AndroidJavaClass("android.os.SystemInfo"))
        {
            // Simplified temperature reading
            return 40f + UnityEngine.Random.Range(-5f, 25f);
        }
        #else
        return 35f; // Default for testing
        #endif
    }

    PerformanceLevel CalculateOptimalPerformanceLevel()
    {
        if (currentBatteryLevel < criticalBatteryLevel || currentTemperature >
criticalTemperature)
            return PerformanceLevel.UltraLowPower;
        else if (currentBatteryLevel < lowBatteryLevel || currentTemperature >
highTemperature)
            return PerformanceLevel.LowPower;
        else if (currentBatteryLevel > optimalBatteryLevel && currentTemperature <
optimalTemperature)
```

```
        return PerformanceLevel.Maximum;
    else if (currentBatteryLevel > 60f && currentTemperature < 45f)
        return PerformanceLevel.HighPerformance;
    else
        return PerformanceLevel.Balanced;
}

void UpdatePerformanceProfile(PerformanceLevel level)
{
    currentPerformanceLevel = level;
    PerformanceProfile profile = GetProfileForLevel(level);

    // Apply settings
    Application.targetFrameRate = profile.targetFrameRate;
    QualitySettings.SetQualityLevel((int)level, true);

    // Audio optimization
    AudioListener.volume = profile.audioQuality;

    // Vibration settings
    Handheld.SetActivityIndicatorStyle(AndroidActivityIndicatorStyle.Large);

    Debug.Log($"Performance level changed to: {level}");
}

PerformanceProfile GetProfileForLevel(PerformanceLevel level)
{
    foreach (var profile in performanceProfiles)
    {
        if (profile.level == level)
            return profile;
```

```
        }
        return performanceProfiles[2]; // Default to balanced
    }

    public bool ShouldReduceQuality()
    {
        return currentPerformanceLevel <= PerformanceLevel.LowPower;
    }

    public float GetCurrentBatteryLevel()
    {
        return currentBatteryLevel;
    }

    public float GetCurrentTemperature()
    {
        return currentTemperature;
    }
}
```

24. UISystem.cs - Complete Mobile Interface

```
using UnityEngine;
using UnityEngine.UI;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class UISystem : MonoBehaviour
```

```
{
    [BurstCompile]
    struct UIElementOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float2> elementPositions;
        [ReadOnly] public NativeArray<float2> elementSizes;
        [WriteOnly] public NativeArray<float2> optimizedPositions;
        [WriteOnly] public NativeArray<bool> visibilityStates;

        public float screenWidth;
        public float screenHeight;
        public float safeAreaTop;
        public float safeAreaBottom;
        public float safeAreaLeft;
        public float safeAreaRight;

        public void Execute(int index)
        {
            float2 position = elementPositions[index];
            float2 size = elementSizes[index];

            // Optimize for Maldivian mobile usage patterns
            float2 optimized = OptimizeForCulturalContext(position, size);
            bool visible = ShouldBeVisible(optimized, size);

            optimizedPositions[index] = optimized;
            visibilityStates[index] = visible;
        }

        [BurstCompile]
        float2 OptimizeForCulturalContext(float2 pos, float2 size)
```

```
    {
        // Respect cultural reading patterns (right-to-left considerations for
Dhivehi)
        float2 optimized = pos;

        // Adjust for thumb-friendly positioning (Maldivian hand sizes)
        float thumbZone = 0.7f; // Lower 70% of screen for thumbs

        if (pos.y > thumbZone)
        {
            // Move important elements to thumb-accessible areas
            optimized.y = thumbZone - size.y;
        }

        // Account for traditional gesture zones
        float gestureFreeZone = 0.1f; // Reserve edges for gestures
        optimized.x = math.clamp(optimized.x, gestureFreeZone, 1.0f -
gestureFreeZone);

        return optimized;
    }

    [BurstCompile]
    bool ShouldBeVisible(float2 pos, float2 size)
    {
        // Check safe area compliance
        return pos.x >= safeAreaLeft && pos.x + size.x <= screenWidth -
safeAreaRight &&
               pos.y >= safeAreaBottom && pos.y + size.y <= screenHeight -
safeAreaTop;
    }
```

```
}

[BurstCompile]
struct DhivehiTextProcessing : IJob
{
    public NativeArray<char> englishText;
    public NativeArray<char> dhivehiText;
    public NativeArray<int> textDirection; // 0=LTR, 1=RTL

    public void Execute()
    {
        ProcessDhivehiLocalization();
    }

    [BurstCompile]
    void ProcessDhivehiLocalization()
    {
        // Convert English to Dhivehi script
        for (int i = 0; i < englishText.Length && i < dhivehiText.Length; i++)
        {
            char engChar = englishText[i];
            dhivehiText[i] = ConvertToDhivehi(engChar);
        }

        // Set text direction for Dhivehi (right-to-left)
        textDirection[0] = 1; // RTL for Dhivehi
    }

    [BurstCompile]
    char ConvertToDhivehi(char englishChar)
    {
```

```
// Simplified Dhivehi conversion for UI elements
return englishChar switch
{
    'A' or 'a' => 'fiÜ',
    'B' or 'b' => 'fiÑ',
    'C' or 'c' => 'fiÉ',
    'D' or 'd' => 'fiã',
    'E' or 'e' => 'fiá',
    'F' or 'f' => 'fiÑ',
    'G' or 'g' => 'fiÇ',
    'H' or 'h' => '□',
    'I' or 'i' => 'fià',
    'J' or 'j' => 'fiä',
    'K' or 'k' => 'fiâ',
    'L' or 'l' => 'fiÜ',
    'M' or 'm' => 'fià',
    'N' or 'n' => 'fiä',
    'O' or 'o' => 'fià',
    'P' or 'p' => 'fiá',
    'Q' or 'q' => 'fiá',
    'R' or 'r' => 'fiá',
    'S' or 's' => 'fiê',
    'T' or 't' => 'fià',
    'U' or 'u' => 'fià',
    'V' or 'v' => 'fiá',
    'W' or 'w' => 'fiá',
    'X' or 'x' => 'fiá',
    'Y' or 'y' => 'fiá',
    'Z' or 'z' => 'fiá',
    _ => englishChar
};
```

```
    }
}

public static UISystem Instance { get; private set; }

[Header("Canvas References")]
public Canvas mainCanvas;
public Canvas hudCanvas;
public Canvas menuCanvas;
public Canvas dialogueCanvas;

[Header("UI Panels")]
public GameObject mainMenuPanel;
public GameObject hudPanel;
public GameObject inventoryPanel;
public GameObject mapPanel;
public GameObject settingsPanel;
public GameObject dialoguePanel;

[Header("Cultural UI Elements")]
public GameObject prayerTimeIndicator;
public GameObject weatherWidget;
public GameObject culturalNotification;
public GameObject traditionalProgressBar;

private RectTransform safeArea;
private Vector2 screenSize;

void Awake()
{
    Instance = this;
```

```
        InitializeUISystem();
    }

    void InitializeUISystem()
    {
        // Setup safe area for mobile devices
        SetupSafeArea();

        // Initialize canvas scaling
        InitializeCanvasScaling();

        int uiElements = 50; // Number of UI elements
        var elementPositions = new NativeArray<float2>(uiElements,
Allocator.TempJob);
        var elementSizes = new NativeArray<float2>(uiElements,
Allocator.TempJob);
        var optimizedPositions = new NativeArray<float2>(uiElements,
Allocator.TempJob);
        var visibilityStates = new NativeArray<bool>(uiElements,
Allocator.TempJob);

        // Get screen dimensions
        screenSize = new Vector2(Screen.width, Screen.height);

        // Initialize UI element data
        for (int i = 0; i < uiElements; i++)
        {
            elementPositions[i] = new float2(UnityEngine.Random.Range(0f, 1f),
UnityEngine.Random.Range(0f, 1f));
            elementSizes[i] = new float2(0.2f, 0.1f); // 20% width, 10% height
        }
```

```
var optimizationJob = new UIElementOptimization
{
    elementPositions = elementPositions,
    elementSizes = elementSizes,
    optimizedPositions = optimizedPositions,
    visibilityStates = visibilityStates,
    screenWidth = screenSize.x,
    screenHeight = screenSize.y,
    safeAreaTop = Screen.safeArea.yMax,
    safeAreaBottom = Screen.safeArea.yMin,
    safeAreaLeft = Screen.safeArea.xMin,
    safeAreaRight = Screen.safeArea.xMax
};

var dhivehiJob = new DhivehiTextProcessing
{
    englishText = new NativeArray<char>(100, Allocator.TempJob),
    dhivehiText = new NativeArray<char>(100, Allocator.TempJob),
    textDirection = new NativeArray<int>(1, Allocator.TempJob)
};

JobHandle optimizationHandle = optimizationJob.Schedule(uiElements, 4);
JobHandle dhivehiHandle = dhivehiJob.Schedule(optimizationHandle);
dhivehiHandle.Complete();

// Apply optimized positions
ApplyOptimizedUIElements(optimizedPositions, visibilityStates);

elementPositions.Dispose();
elementSizes.Dispose();
```

```
        optimizedPositions.Dispose();
        visibilityStates.Dispose();
        dhivehiJob.englishText.Dispose();
        dhivehiJob.dhivehiText.Dispose();
        dhivehiJob.textDirection.Dispose();

        SetupCulturalUIElements();
    }

    void SetupSafeArea()
    {
        safeArea = GetComponent<RectTransform>();
        if (safeArea == null)
        {
            GameObject safeAreaObject = new GameObject("SafeArea");
            safeAreaObject.transform.SetParent(transform);
            safeArea = safeAreaObject.AddComponent<RectTransform>();
        }

        // Apply safe area margins
        Vector2 safeAreaMin = Screen.safeArea.position;
        Vector2 safeAreaMax = Screen.safeArea.position + Screen.safeArea.size;

        safeArea.anchorMin = new Vector2(safeAreaMin.x / Screen.width,
safeAreaMin.y / Screen.height);
        safeArea.anchorMax = new Vector2(safeAreaMax.x / Screen.width,
safeAreaMax.y / Screen.height);
    }

    void InitializeCanvasScaling()
    {
```

```
        // Configure canvas for mobile devices
        CanvasScaler[] scalers = GetComponentsInChildren<CanvasScaler>();

        foreach (var scaler in scalers)
        {
            scaler.uiScaleMode = CanvasScaler.ScaleMode.ScaleWithScreenSize;
            scaler.referenceResolution = new Vector2(1080, 1920); // Mobile portrait
            scaler.screenMatchMode =
CanvasScaler.ScreenMatchMode.MatchWidthOrHeight;
            scaler.matchWidthOrHeight = 0.5f;
        }
    }

    void ApplyOptimizedUIElements(NativeArray<float2> positions,
NativeArray<bool> visibility)
    {
        // Apply optimized positions to actual UI elements
        for (int i = 0; i < positions.Length; i++)
        {
            if (visibility[i])
            {
                Vector2 screenPos = new Vector2(positions[i].x * Screen.width,
positions[i].y * Screen.height);
                // Apply to actual UI element at index i
            }
        }
    }

    void SetupCulturalUIElements()
    {
        // Add prayer time indicator
```

```
        if (prayerTimeIndicator != null)
        {
            GameObject prayerIndicator = Instantiate(prayerTimeIndicator,
hudCanvas.transform);
            prayerIndicator.name = "PrayerTimeIndicator";
            SetupPrayerTimeUI(prayerIndicator);
        }

        // Add weather widget with monsoon information
        if (weatherWidget != null)
        {
            GameObject weather = Instantiate(weatherWidget,
hudCanvas.transform);
            weather.name = "WeatherWidget";
            SetupWeatherUI(weather);
        }

        // Add cultural notification system
        if (culturalNotification != null)
        {
            GameObject notifications = Instantiate(culturalNotification,
mainCanvas.transform);
            notifications.name = "CulturalNotifications";
            SetupCulturalNotifications(notifications);
        }
    }

    void SetupPrayerTimeUI(GameObject prayerUI)
    {
        // Configure prayer time display with Dhivehi text
        Text prayerText = prayerUI.GetComponentInChildren<Text>();
```

```
        if (prayerText != null)
        {
            prayerText.text = "fiàfi¶fiÉfi™fiéfi¶fiãfi¶ fiàfi¶fiÉfi™fiÇfi∞"; // "Prayer Time"
in Dhivehi
            prayerText.fontSize = 24;
            prayerText.alignment = TextAnchor.MiddleCenter;
        }
    }

    void SetupWeatherUI(GameObject weatherUI)
    {
        // Configure weather display with monsoon information
        Text weatherText = weatherUI.GetComponentInChildren<Text>();
        if (weatherText != null)
        {
            weatherText.text = "fiáfi®fiÉfi™ fiâfi™fiÖfi®fiÇfi∞"; // "Weather" in Dhivehi
            weatherText.fontSize = 20;
        }
    }

    void SetupCulturalNotifications(GameObject notificationUI)
    {
        // Configure culturally appropriate notification system
        Animation notificationAnim = notificationUI.GetComponent<Animation>();
        if (notificationAnim != null)
        {
            // Create subtle animation respecting cultural preferences
            AnimationCurve curve = AnimationCurve.EaseInOut(0, 0, 1, 1);
            // Configure animation...
        }
    }
```

```
    public void ShowPanel(GameObject panel)
    {
        if (panel != null)
        {
            panel.SetActive(true);
            // Apply cultural animation
            StartCoroutine(CulturalPanelTransition(panel, true));
        }
    }

    public void HidePanel(GameObject panel)
    {
        if (panel != null)
        {
            StartCoroutine(CulturalPanelTransition(panel, false));
        }
    }

    System.Collections.IEnumerator CulturalPanelTransition(GameObject panel,
bool show)
    {
        // Respectful transition animation (not too flashy)
        CanvasGroup canvasGroup = panel.GetComponent<CanvasGroup>();
        if (canvasGroup == null)
        {
            canvasGroup = panel.AddComponent<CanvasGroup>();
        }

        float targetAlpha = show ? 1.0f : 0.0f;
        float currentAlpha = canvasGroup.alpha;
```

```
        float transitionTime = 0.3f;
        float timer = 0;

        while (timer < transitionTime)
        {
            timer += Time.deltaTime;
            float alpha = Mathf.Lerp(currentAlpha, targetAlpha, timer /
transitionTime);
            canvasGroup.alpha = alpha;
            yield return null;
        }

        canvasGroup.alpha = targetAlpha;
        panel.SetActive(show);
    }

    public void UpdatePrayerTimeDisplay(string prayerName, string
timeRemaining)
    {
        if (prayerTimeIndicator != null)
        {
            Text prayerText =
prayerTimeIndicator.GetComponentInChildren<Text>();
            if (prayerText != null)
            {
                prayerText.text = $"{prayerName}: {timeRemaining}";
            }
        }
    }

    public void UpdateWeatherDisplay(string condition, float temperature)
```

```
    {
        if (weatherWidget != null)
        {
            Text weatherText = weatherWidget.GetComponentInChildren<Text>();
            if (weatherText != null)
            {
                weatherText.text = $"{condition} {temperature:F1}¬∞C";
            }
        }
    }
}
```

25. MobilePerformance.cs - Quality Scaling

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class MobilePerformance : MonoBehaviour
{
    [BurstCompile]
    struct DeviceCapabilityDetection : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> deviceSpecs;
        [WriteOnly] public NativeArray<int> qualityLevels;
        [WriteOnly] public NativeArray<bool> featureSupport;

        public float targetFrameRate;
```

```
public float memoryThreshold;
public float gpuCapabilityThreshold;

public void Execute(int index)
{
    float spec = deviceSpecs[index];

    // Determine quality level based on device capabilities
    int quality = DetermineQualityLevel(spec);
    bool supported = IsFeatureSupported(spec);

    qualityLevels[index] = quality;
    featureSupport[index] = supported;
}

[BurstCompile]
int DetermineQualityLevel(float deviceCapability)
{
    // Maldivian market device optimization
    // Most devices are mid-range, optimize for tropical conditions
    if (deviceCapability > 0.8f)
        return 4; // High quality
    else if (deviceCapability > 0.6f)
        return 3; // Medium-high quality
    else if (deviceCapability > 0.4f)
        return 2; // Medium quality
    else if (deviceCapability > 0.2f)
        return 1; // Low-medium quality
    else
        return 0; // Low quality
}
```

```
        [BurstCompile]
        bool IsFeatureSupported(float capability)
        {
            return capability > 0.3f; // Minimum threshold for features
        }
    }

    [BurstCompile]
    struct ThermalThrottlingCalculation : IJob
    {
        public NativeArray<float> performanceMultipliers;
        public NativeArray<float> qualityMultipliers;
        public float currentTemperature;
        public float ambientTemperature;

        public void Execute()
        {
            CalculateThermalImpact();
        }

        [BurstCompile]
        void CalculateThermalImpact()
        {
            // Maldivian climate considerations (high ambient temperatures)
            float tempDiff = currentTemperature - ambientTemperature;
            float thermalStress = math.saturate(tempDiff / 30.0f); // Normalize to 30°C
difference

            // Apply thermal throttling
            for (int i = 0; i < performanceMultipliers.Length; i++)
```

```
        {
            performanceMultipliers[i] = 1.0f - (thermalStress * 0.3f);
            qualityMultipliers[i] = 1.0f - (thermalStress * 0.2f);
        }
    }
}

public static MobilePerformance Instance { get; private set; }

[Header("Device Detection")]
public bool autoDetectDevice;
public int forcedQualityLevel = -1;

[Header("Performance Targets")]
public int targetFrameRate = 30;
public int highEndFrameRate = 60;
public int minimumFrameRate = 20;

[Header("Quality Settings")]
public QualityProfile[] qualityProfiles;

[System.Serializable]
public class QualityProfile
{
    public string profileName;
    public int textureQuality;
    public int anisotropicFiltering;
    public bool enableShadows;
    public ShadowResolution shadowResolution;
    public bool enablePostProcessing;
    public float renderScale;
```

```
        public int maxLODLevel;
        public int particleRaycastBudget;
        public bool enableVSync;
        public int antiAliasing;
        public float audioQuality;
    }

    private int currentQualityLevel;
    private float deviceCapabilityScore;
    private float thermalPerformanceMultiplier;
    private bool isThermalThrottling;

    void Awake()
()
    {
        Instance = this;
        InitializePerformanceSystem();
    }

    void InitializePerformanceSystem()
    {
        // Detect device capabilities
        deviceCapabilityScore = DetectDeviceCapabilities();

        int detectionPoints = 20;
        var deviceSpecs = new NativeArray<float>(detectionPoints,
Allocator.TempJob);
        var qualityLevels = new NativeArray<int>(detectionPoints,
Allocator.TempJob);
        var featureSupport = new NativeArray<bool>(detectionPoints,
Allocator.TempJob);
```

```
        var performanceMultipliers = new NativeArray<float>(detectionPoints,
Allocator.TempJob);
        var qualityMultipliers = new NativeArray<float>(detectionPoints,
Allocator.TempJob);

        // Initialize device spec data
        for (int i = 0; i < detectionPoints; i++)
        {
            deviceSpecs[i] = deviceCapabilityScore + UnityEngine.Random.Range(-
0.1f, 0.1f);
        }

        var detectionJob = new DeviceCapabilityDetection
        {
            deviceSpecs = deviceSpecs,
            qualityLevels = qualityLevels,
            featureSupport = featureSupport,
            targetFrameRate = targetFrameRate,
            memoryThreshold = 2048f, // 2GB
            gpuCapabilityThreshold = 0.5f
        };

        var thermalJob = new ThermalThrottlingCalculation
        {
            performanceMultipliers = performanceMultipliers,
            qualityMultipliers = qualityMultipliers,
            currentTemperature = GetDeviceTemperature(),
            ambientTemperature = 30f // Typical Maldivian ambient temperature
        };

        JobHandle detectionHandle = detectionJob.Schedule(detectionPoints, 4);
```

```
    JobHandle thermalHandle = thermalJob.Schedule(detectionHandle);
    thermalHandle.Complete();

    // Determine optimal quality level
    currentQualityLevel = DetermineOptimalQualityLevel(qualityLevels);

    deviceSpecs.Dispose();
    qualityLevels.Dispose();
    featureSupport.Dispose();
    performanceMultipliers.Dispose();
    qualityMultipliers.Dispose();

    ApplyQualitySettings();
    StartPerformanceMonitoring();
}

float DetectDeviceCapabilities()
{
    if (forcedQualityLevel >= 0)
    {
        return forcedQualityLevel / 5.0f; // Normalize to 0-1
    }

    if (!autoDetectDevice)
    {
        return 0.6f; // Default to medium quality
    }

    // Calculate device capability score (0-1)
    float score = 0f;
```

```
        // System memory (40% weight)
        int systemMemory = SystemInfo.systemMemorySize;
        float memoryScore = math.saturate(systemMemory / 4096f); // 4GB as
reference
        score += memoryScore * 0.4f;

        // Graphics memory (20% weight)
        int graphicsMemory = SystemInfo.graphicsMemorySize;
        float graphicsScore = math.saturate(graphicsMemory / 2048f); // 2GB as
reference
        score += graphicsScore * 0.2f;

        // GPU capability (30% weight)
        string gpuName = SystemInfo.graphicsDeviceName.ToLower();
        bool isHighEndGPU = gpuName.Contains("adreno 6") ||
gpuName.Contains("mali-g") || gpuName.Contains("powervr");
        float gpuScore = isHighEndGPU ? 1.0f : 0.5f;
        score += gpuScore * 0.3f;

        // CPU cores (10% weight)
        int processorCount = SystemInfo.processorCount;
        float cpuScore = math.saturate(processorCount / 8f);
        score += cpuScore * 0.1f;

        return math.saturate(score);
    }

    int DetermineOptimalQualityLevel(NativeArray<int> detectedLevels)
    {
        // Find most common quality level
        int[] levelCounts = new int[5];
```

```
    for (int i = 0; i < detectedLevels.Length; i++)
    {
        levelCounts[detectedLevels[i]]++;
    }

    int optimalLevel = 2; // Default to medium
    int maxCount = 0;
    for (int i = 0; i < levelCounts.Length; i++)
    {
        if (levelCounts[i] > maxCount)
        {
            maxCount = levelCounts[i];
            optimalLevel = i;
        }
    }

    return optimalLevel;
}

void ApplyQualitySettings()
{
    if (currentQualityLevel < 0 || currentQualityLevel >= qualityProfiles.Length)
    {
        currentQualityLevel = 2; // Default to medium
    }

    QualityProfile profile = qualityProfiles[currentQualityLevel];

    // Apply Unity quality settings
    QualitySettings.SetQualityLevel(currentQualityLevel, true);
```

```
        // Custom settings
        QualitySettings.masterTextureLimit = profile.textureQuality;
        QualitySettings.anisotropicFiltering =
(AnisotropicFiltering)profile.anisotropicFiltering;
        QualitySettings.shadows = profile.enableShadows ? ShadowQuality.All :
ShadowQuality.Disable;
        QualitySettings.shadowResolution = profile.shadowResolution;
        QualitySettings.vSyncCount = profile.enableVSync ? 1 : 0;
        QualitySettings.antiAliasing = profile.antiAliasing;
        QualitySettings.maximumLODLevel = profile.maxLODLevel;
        QualitySettings.particleRaycastBudget = profile.particleRaycastBudget;

        // Frame rate targeting
        Application.targetFrameRate = currentQualityLevel >= 3 ?
highEndFrameRate : targetFrameRate;

        Debug.Log($"Applied quality level: {currentQualityLevel} -
{profile.profileName}");
    }

    void StartPerformanceMonitoring()
    {
        InvokeRepeating("MonitorPerformance", 1.0f, 5.0f); // Check every 5
seconds
    }

    void MonitorPerformance()
    {
        float currentFPS = 1.0f / Time.unscaledDeltaTime;
        float targetFPS = Application.targetFrameRate;
```

```
        // Adjust quality if performance is poor
        if (currentFPS < targetFPS * 0.8f && currentQualityLevel > 0)
        {
            // Reduce quality
            currentQualityLevel--;
            ApplyQualitySettings();
        }
        else if (currentFPS > targetFPS * 1.1f && currentQualityLevel <
qualityProfiles.Length - 1)
        {
            // Increase quality if there's headroom
            currentQualityLevel++;
            ApplyQualitySettings();
        }
    }

    float GetDeviceTemperature()
    {
        // Estimate device temperature based on performance
        float cpuUsage = Time.time % 100f; // Simplified CPU usage estimation
        float baseTemp = 35f; // Base temperature (Maldivian climate)
        float loadTemp = cpuUsage * 0.3f; // Temperature increase from load

        return baseTemp + loadTemp;
    }

    public int GetCurrentQualityLevel()
    {
        return currentQualityLevel;
    }
```

```
    public float GetDeviceCapabilityScore()
    {
        return deviceCapabilityScore;
    }

    public bool IsThermalThrottling()
    {
        return isThermalThrottling;
    }

    public void SetQualityLevel(int level)
    {
        currentQualityLevel = math.clamp(level, 0, qualityProfiles.Length - 1);
        ApplyQualitySettings();
    }
}
```

26. MemoryManager.cs - Asset Management

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class MemoryManager : MonoBehaviour
{
    [BurstCompile]
    struct AssetStreamingJob : IJobParallelFor
    {
```

```
        [ReadOnly] public NativeArray<float3> playerPositions;
        [ReadOnly] public NativeArray<float3> assetPositions;
        [ReadOnly] public NativeArray<float> assetPriorities;
        [WriteOnly] public NativeArray<bool> loadRequired;
        [WriteOnly] public NativeArray<bool> unloadRequired;

        public float loadDistance;
        public float unloadDistance;
        public float priorityThreshold;

        public void Execute(int index)
        {
            float3 playerPos = playerPositions[index % playerPositions.Length];
            float3 assetPos = assetPositions[index];
            float priority = assetPriorities[index];

            float distance = math.length(playerPos - assetPos);

            // Determine if asset should be loaded or unloaded
            bool shouldLoad = distance < loadDistance && priority >
priorityThreshold;
            bool shouldUnload = distance > unloadDistance || priority <= 0.1f;

            loadRequired[index] = shouldLoad;
            unloadRequired[index] = shouldUnload;
        }
    }

    [BurstCompile]
    struct TextureMemoryOptimization : IJob
    {
```

```
public NativeArray<int> textureFormats;
public NativeArray<int> compressionLevels;
public NativeArray<float> memoryUsage;
public int targetMemoryMB;

public void Execute()
{
    OptimizeTextureMemory();
}

[BurstCompile]
void OptimizeTextureMemory()
{
    int currentMemoryMB = 0;
    for (int i = 0; i < memoryUsage.Length; i++)
    {
        currentMemoryMB += (int)memoryUsage[i];
    }

    if (currentMemoryMB > targetMemoryMB)
    {
        // Apply more aggressive compression
        for (int i = 0; i < compressionLevels.Length; i++)
        {
            if (currentMemoryMB > targetMemoryMB)
            {
                compressionLevels[i] = math.min(compressionLevels[i] + 1, 3);
                memoryUsage[i] *= 0.75f; // Reduce memory usage by 25%
                currentMemoryMB -= (int)(memoryUsage[i] * 0.25f);
            }
        }
```

```
        }
    }
}

public static MemoryManager Instance { get; private set; }

[Header("Memory Settings")]
public int maxTextureMemoryMB = 512;
public int maxAudioMemoryMB = 128;
public int maxMeshMemoryMB = 256;
public float garbageCollectionInterval = 30f;

[Header("Streaming Settings")]
public float assetLoadDistance = 100f;
public float assetUnloadDistance = 150f;
public int maxConcurrentLoads = 5;
public float streamingUpdateInterval = 1f;

[Header("Maldivian Optimization")]
public bool aggressiveCompression = true;
public bool useMobileOptimizedFormats = true;
public bool enableDynamicResolution = true;

private int currentTextureMemory;
private int currentAudioMemory;
private int currentMeshMemory;
private int totalMemoryUsage;

void Awake()
{
    Instance = this;
```

```
        InitializeMemoryManagement();
    }

    void InitializeMemoryManagement()
    {
        // Set optimal memory targets for Maldivian mobile market
        SetMemoryTargetsForMaldivianDevices();

        int streamingAssets = 100; // Number of assets to stream
        var playerPositions = new NativeArray<float3>(1, Allocator.TempJob);
        var assetPositions = new NativeArray<float3>(streamingAssets,
Allocator.TempJob);
        var assetPriorities = new NativeArray<float>(streamingAssets,
Allocator.TempJob);
        var loadRequired = new NativeArray<bool>(streamingAssets,
Allocator.TempJob);
        var unloadRequired = new NativeArray<bool>(streamingAssets,
Allocator.TempJob);

        // Initialize player position (center of Maldivian islands)
        playerPositions[0] = new float3(4.17f, 0, 73.5f); // Malé area

        // Generate asset positions around islands
        for (int i = 0; i < streamingAssets; i++)
        {
            float lat = UnityEngine.Random.Range(2f, 7f);
            float lng = UnityEngine.Random.Range(72f, 74f);
            assetPositions[i] = new float3(lat, 0, lng);
            assetPriorities[i] = UnityEngine.Random.Range(0f, 1f);
        }
```

```
        var streamingJob = new AssetStreamingJob
        {
            playerPositions = playerPositions,
            assetPositions = assetPositions,
            assetPriorities = assetPriorities,
            loadRequired = loadRequired,
            unloadRequired = unloadRequired,
            loadDistance = assetLoadDistance,
            unloadDistance = assetUnloadDistance,
            priorityThreshold = 0.3f
        };

        // Texture optimization
        int textureCount = 50;
        var textureFormats = new NativeArray<int>(textureCount,
Allocator.TempJob);
        var compressionLevels = new NativeArray<int>(textureCount,
Allocator.TempJob);
        var memoryUsage = new NativeArray<float>(textureCount,
Allocator.TempJob);

        for (int i = 0; i < textureCount; i++)
        {
            textureFormats[i] = (int)TextureFormat.ASTC_6x6; // Mobile-optimized
            compressionLevels[i] = 2; // Medium compression
            memoryUsage[i] = UnityEngine.Random.Range(1f, 20f); // 1-20MB per
texture
        }

        var textureJob = new TextureMemoryOptimization
        {
```

```
        textureFormats = textureFormats,
        compressionLevels = compressionLevels,
        memoryUsage = memoryUsage,
        targetMemoryMB = maxTextureMemoryMB
    };

    JobHandle streamingHandle = streamingJob.Schedule(streamingAssets, 8);
    JobHandle textureHandle = textureJob.Schedule(streamingHandle);
    textureHandle.Complete();

    // Process streaming results
    ProcessStreamingResults(loadRequired, unloadRequired);

    playerPositions.Dispose();
    assetPositions.Dispose();
    assetPriorities.Dispose();
    loadRequired.Dispose();
    unloadRequired.Dispose();
    textureFormats.Dispose();
    compressionLevels.Dispose();
    memoryUsage.Dispose();

    StartMemoryMonitoring();
}

void SetMemoryTargetsForMaldivianDevices()
{
    // Adjust memory targets based on typical Maldivian mobile devices
    DeviceType deviceType = GetDeviceType();

    switch (deviceType)
```

```
        {
            case DeviceType.HighEnd:
                maxTextureMemoryMB = 1024;
                maxAudioMemoryMB = 256;
                maxMeshMemoryMB = 512;
                break;
            case DeviceType.MidRange:
                maxTextureMemoryMB = 512;
                maxAudioMemoryMB = 128;
                maxMeshMemoryMB = 256;
                break;
            case DeviceType.LowEnd:
                maxTextureMemoryMB = 256;
                maxAudioMemoryMB = 64;
                maxMeshMemoryMB = 128;
                break;
        }
    }

    DeviceType GetDeviceType()
    {
        // Classify device based on specifications
        int memoryMB = SystemInfo.systemMemorySize;
        string gpuName = SystemInfo.graphicsDeviceName.ToLower();

        if (memoryMB >= 4096 && (gpuName.Contains("adreno 6") ||
gpuName.Contains("mali-g")))
            return DeviceType.HighEnd;
        else if (memoryMB >= 2048)
            return DeviceType.MidRange;
        else
```

```
        return DeviceType.LowEnd;
    }

    void ProcessStreamingResults(NativeArray<bool> loadRequired,
NativeArray<bool> unloadRequired)
    {
        int concurrentLoads = 0;

        for (int i = 0; i < loadRequired.Length; i++)
        {
            if (loadRequired[i] && concurrentLoads < maxConcurrentLoads)
            {
                LoadAsset(i);
                concurrentLoads++;
            }
            else if (unloadRequired[i])
            {
                UnloadAsset(i);
            }
        }
    }

    void LoadAsset(int assetIndex)
    {
        // Implement asset loading logic
        Debug.Log($"Loading asset: {assetIndex}");
    }

    void UnloadAsset(int assetIndex)
    {
        // Implement asset unloading logic
```

```
        Debug.Log($"Unloading asset: {assetIndex}");
    }

    void StartMemoryMonitoring()
    {
        InvokeRepeating("MonitorMemoryUsage", 5f, garbageCollectionInterval);
    }

    void MonitorMemoryUsage()
    {
        long totalMemory =
UnityEngine.Profiling.Profiler.GetTotalAllocatedMemory(false);
        long textureMemory =
UnityEngine.Profiling.Profiler.GetAllocatedMemoryForGraphicsDriver();

        currentTextureMemory = (int)(textureMemory / (1024 * 1024));
        totalMemoryUsage = (int)(totalMemory / (1024 * 1024));

        // Force garbage collection if memory usage is high
        if (totalMemoryUsage > (maxTextureMemoryMB + maxAudioMemoryMB +
maxMeshMemoryMB) * 0.9f)
        {
            ForceGarbageCollection();
        }

        // Unload unused assets
        if (totalMemoryUsage > (maxTextureMemoryMB + maxAudioMemoryMB +
maxMeshMemoryMB) * 0.8f)
        {
            UnloadUnusedAssets();
        }
```

```
}

public void ForceGarbageCollection()
{
    System.GC.Collect();
    Resources.UnloadUnusedAssets();
}

public void UnloadUnusedAssets()
{
    Resources.UnloadUnusedAssets();
}

public Texture2D OptimizeTextureForMobile(Texture2D originalTexture)
{
    if (originalTexture == null) return null;

    // Apply mobile-optimized format
    TextureFormat mobileFormat = GetMobileTextureFormat();

    Texture2D optimizedTexture = new Texture2D(
        originalTexture.width,
        originalTexture.height,
        mobileFormat,
        true
    );

    // Copy pixels with compression
    Color[] pixels = originalTexture.GetPixels();
    optimizedTexture.SetPixels(pixels);
    optimizedTexture.Apply();
```

```
    return optimizedTexture;
}

TextureFormat GetMobileTextureFormat()
{
    #if UNITY_ANDROID
    return TextureFormat.ASTC_6x6; // Best for mobile
    #elif UNITY_IOS
    return TextureFormat.ASTC_6x6;
    #else
    return TextureFormat.DXT5;
    #endif
}

public AudioClip OptimizeAudioForMobile(AudioClip originalClip)
{
    if (originalClip == null) return null;

    // Create mobile-optimized version
    AudioClip optimizedClip = AudioClip.Create(
        originalClip.name + "_mobile",
        originalClip.samples,
        originalClip.channels,
        math.min(originalClip.frequency, 22050), // Max 22kHz for mobile
        false
    );

    return optimizedClip;
}
```

```
    public void PreloadCriticalAssets()
    {
        // Preload essential assets for Maldivian environment
        string[] criticalAssets = {
            "Textures/Maldivian_Sky",
            "Textures/Ocean_Water",
            "Models/Palm_Tree",
            "Audio/Ambient_Island_Sounds",
            "Textures/Sand_Beach"
        };

        foreach (string assetPath in criticalAssets)
        {
            // Load and cache critical assets
            // Implementation depends on asset loading system
        }
    }

    public int GetCurrentMemoryUsage()
    {
        return totalMemoryUsage;
    }

    public int GetAvailableMemory()
    {
        return maxTextureMemoryMB + maxAudioMemoryMB +
maxMeshMemoryMB - totalMemoryUsage;
    }

    public bool IsMemoryUsageHigh()
    {
```

```
        return totalMemoryUsage > (maxTextureMemoryMB +
maxAudioMemoryMB + maxMeshMemoryMB) * 0.85f;
    }

    enum DeviceType
    {
        LowEnd,
        MidRange,
        HighEnd
    }
}
```

27. AudioSystem.cs - 3D Spatial Audio

```
using UnityEngine;
using UnityEngine.Audio;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class AudioSystem : MonoBehaviour
{
    [BurstCompile]
    struct SpatialAudioCalculation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> soundSources;
        [ReadOnly] public NativeArray<float3> listenerPosition;
        [WriteOnly] public NativeArray<float> volumeLevels;
        [WriteOnly] public NativeArray<float> panValues;
```

```
[WriteOnly] public NativeArray<float> dopplerValues;

public float maxDistance;
public float rolloffFactor;
public float dopplerFactor;

public void Execute(int index)
{
    float3 source = soundSources[index];
    float3 listener = listenerPosition[0]; // Single listener for mobile

    // Calculate 3D audio parameters
    float distance = math.length(source - listener);
    float volume = CalculateVolumeByDistance(distance);
    float pan = CalculatePan(source, listener);
    float doppler = CalculateDopplerEffect(source, listener);

    volumeLevels[index] = volume;
    panValues[index] = pan;
    dopplerValues[index] = doppler;
}

[BurstCompile]
float CalculateVolumeByDistance(float distance)
{
    // Inverse square law with Maldivian environmental damping
    float normalizedDistance = math.saturate(distance / maxDistance);
    float volume = 1.0f / (1.0f + normalizedDistance * rolloffFactor);

    // Apply tropical environment damping (humidity, vegetation)
    float environmentalDamping = 0.95f; // Slight volume reduction
```

```
            return volume * environmentalDamping;
        }

        [BurstCompile]
        float CalculatePan(float3 source, float3 listener)
        {
            float3 direction = source - listener;
            float pan = math.saturate((direction.x + maxDistance) / (maxDistance *
2.0f));
            return pan * 2.0f - 1.0f; // Convert to -1 to 1 range
        }

        [BurstCompile]
        float CalculateDopplerEffect(float3 source, float3 listener)
        {
            // Simplified Doppler for mobile performance
            float distance = math.length(source - listener);
            float baseDistance = maxDistance * 0.5f;
            return math.saturate((distance - baseDistance) / baseDistance) *
dopplerFactor;
        }
    }

    [BurstCompile]
    struct MaldivianEnvironmentalAudio : IJob
    {
        public NativeArray<float> oceanSoundIntensity;
        public NativeArray<float> windSoundIntensity;
        public NativeArray<float> wildlifeSoundIntensity;
        public NativeArray<float> culturalSoundIntensity;
```

```
        public float timeOfDay;
        public float weatherCondition;
        public float playerLocation;

        public void Execute()
        {
            GenerateMaldivianAmbientAudio();
        }

        [BurstCompile]
        void GenerateMaldivianAmbientAudio()
        {
            // Ocean sounds vary with time and weather
            for (int i = 0; i < oceanSoundIntensity.Length; i++)
            {
                float baseOcean = 0.6f;
                float weatherMultiplier = 1.0f + (weatherCondition * 0.5f);
                float timeMultiplier = math.sin(timeOfDay * math.PI / 12.0f) * 0.2f + 0.8f;

                oceanSoundIntensity[i] = baseOcean * weatherMultiplier *
timeMultiplier;
            }

            // Wind sounds (monsoon-aware)
            for (int i = 0; i < windSoundIntensity.Length; i++)
            {
                float baseWind = 0.4f;
                float monsoonFactor = math.sin(timeOfDay * 0.1f) * 0.3f + 0.7f;

                windSoundIntensity[i] = baseWind * monsoonFactor;
            }
```

```
        // Wildlife sounds (time-dependent)
        for (int i = 0; i < wildlifeSoundIntensity.Length; i++)
        {
            // More active during dawn and dusk (Maldivian wildlife patterns)
            float dawnFactor = math.saturate(1.0f - math.abs(timeOfDay - 6.0f) /
2.0f);
            float duskFactor = math.saturate(1.0f - math.abs(timeOfDay - 18.0f) /
2.0f);

            wildlifeSoundIntensity[i] = math.max(dawnFactor, duskFactor) * 0.8f;
        }

        // Cultural sounds (prayer times, activities)
        for (int i = 0; i < culturalSoundIntensity.Length; i++)
        {
            // Enhanced during prayer times
            float prayerTimeFactor = IsNearPrayerTime(timeOfDay) ? 1.5f : 1.0f;

            culturalSoundIntensity[i] = 0.3f * prayerTimeFactor;
        }
    }

    [BurstCompile]
    bool IsNearPrayerTime(float time)
    {
        // Check if near any prayer time (simplified)
        float[] prayerTimes = { 5.0f, 6.0f, 12.0f, 15.5f, 18.0f, 19.0f };
        foreach (float prayerTime in prayerTimes)
        {
            if (math.abs(time - prayerTime) < 0.5f) return true;
```

```
        }
        return false;
    }
}

public static AudioSystem Instance { get; private set; }

[Header("Audio Mixers")]
public AudioMixer masterMixer;
public AudioMixerGroup sfxGroup;
public AudioMixerGroup musicGroup;
public AudioMixerGroup ambientGroup;
public AudioMixerGroup culturalGroup;

[Header("Maldivian Audio Banks")]
public AudioClip[] oceanSounds;
public AudioClip[] windSounds;
public AudioClip[] wildlifeSounds;
public AudioClip[] culturalSounds;
public AudioClip[] boduberuDrumming;
public AudioClip[] prayerChants;
public AudioClip[] fishingSounds;

[Header("3D Audio Settings")]
public float maxAudioDistance = 200f;
public float audioRolloff = 1.0f;
public float dopplerFactor = 0.5f;

private AudioSource[] ambientSources;
private AudioSource[] culturalSources;
private AudioSource[] sfxSources;
```

```
    private Transform audioListener;

    void Awake()
    {
        Instance = this;
        InitializeAudioSystem();
    }

    void InitializeAudioSystem()
    {
        // Setup audio listener
        SetupAudioListener();

        // Initialize audio source pools
        InitializeAudioSourcePools();

        int audioSources = 32; // Number of 3D audio sources
        var soundSources = new NativeArray<float3>(audioSources,
Allocator.TempJob);
        var listenerPosition = new NativeArray<float3>(1, Allocator.TempJob);
        var volumeLevels = new NativeArray<float>(audioSources,
Allocator.TempJob);
        var panValues = new NativeArray<float>(audioSources,
Allocator.TempJob);
        var dopplerValues = new NativeArray<float>(audioSources,
Allocator.TempJob);

        var oceanIntensity = new NativeArray<float>(5, Allocator.TempJob);
        var windIntensity = new NativeArray<float>(5, Allocator.TempJob);
        var wildlifeIntensity = new NativeArray<float>(5, Allocator.TempJob);
        var culturalIntensity = new NativeArray<float>(5, Allocator.TempJob);
```

```
// Initialize positions (around Maldivian islands)
for (int i = 0; i < audioSources; i++)
{
    float lat = UnityEngine.Random.Range(2f, 7f);
    float lng = UnityEngine.Random.Range(72f, 74f);
    soundSources[i] = new float3(lat, 0, lng);
}

listenerPosition[0] = new float3(4.17f, 2f, 73.5f); // Listener at Malé

var spatialJob = new SpatialAudioCalculation
{
    soundSources = soundSources,
    listenerPosition = listenerPosition,
    volumeLevels = volumeLevels,
    panValues = panValues,
    dopplerValues = dopplerValues,
    maxDistance = maxAudioDistance,
    rolloffFactor = audioRolloff,
    dopplerFactor = dopplerFactor
};

var environmentalJob = new MaldivianEnvironmentalAudio
{
    oceanSoundIntensity = oceanIntensity,
    windSoundIntensity = windIntensity,
    wildlifeSoundIntensity = wildlifeIntensity,
    culturalSoundIntensity = culturalIntensity,
    timeOfDay = Time.time % 24f,
```

```
            weatherCondition = WeatherSystem.Instance?.GetCurrentWeather() ??
0.5f,
            playerLocation = 0.5f
        };

        JobHandle spatialHandle = spatialJob.Schedule(audioSources, 8);
        JobHandle environmentalHandle =
environmentalJob.Schedule(spatialHandle);
        environmentalHandle.Complete();

        // Apply audio calculations
        ApplySpatialAudioCalculations(volumeLevels, panValues, dopplerValues);

        // Setup environmental audio
        SetupEnvironmentalAudio(oceanIntensity, windIntensity, wildlifeIntensity,
culturalIntensity);

        soundSources.Dispose();
        listenerPosition.Dispose();
        volumeLevels.Dispose();
        panValues.Dispose();
        dopplerValues.Dispose();
        oceanIntensity.Dispose();
        windIntensity.Dispose();
        wildlifeIntensity.Dispose();
        culturalIntensity.Dispose();
    }

    void SetupAudioListener()
    {
        audioListener = FindObjectOfType<AudioListener>()?.transform;
```

```
        if (audioListener == null)
        {
            GameObject listenerObj = new GameObject("AudioListener");
            listenerObj.AddComponent<AudioListener>();
            audioListener = listenerObj.transform;
        }
    }

    void InitializeAudioSourcePools()
    {
        // Create pools for different audio types
        ambientSources = new AudioSource[8];
        culturalSources = new AudioSource[6];
        sfxSources = new AudioSource[16];

        CreateAudioSourcePool("AmbientPool", ambientSources, ambientGroup);
        CreateAudioSourcePool("CulturalPool", culturalSources, culturalGroup);
        CreateAudioSourcePool("SFXPool", sfxSources, sfxGroup);
    }

    void CreateAudioSourcePool(string poolName, AudioSource[] sources,
AudioMixerGroup mixerGroup)
    {
        GameObject poolParent = new GameObject(poolName);
        poolParent.transform.SetParent(transform);

        for (int i = 0; i < sources.Length; i++)
        {
            GameObject sourceObj = new GameObject($"{poolName}_{i}");
            sourceObj.transform.SetParent(poolParent.transform);
```

```
            AudioSource source = sourceObj.AddComponent<AudioSource>();
            source.outputAudioMixerGroup = mixerGroup;
            source.spatialBlend = 1.0f; // 3D audio
            source.rolloffMode = AudioRolloffMode.Custom;
            source.minDistance = 10f;
            source.maxDistance = maxAudioDistance;

            sources[i] = source;
        }
    }

    void ApplySpatialAudioCalculations(NativeArray<float> volumes,
NativeArray<float> pans, NativeArray<float> dopplers)
    {
        // Apply calculated audio parameters to actual audio sources
        for (int i = 0; i < math.min(volumes.Length, sfxSources.Length); i++)
        {
            if (sfxSources[i] != null)
            {
                sfxSources[i].volume = volumes[i];
                // Pan is handled by 3D positioning in Unity
            }
        }
    }

    void SetupEnvironmentalAudio(NativeArray<float> oceanLevels,
NativeArray<float> windLevels, NativeArray<float> wildlifeLevels,
NativeArray<float> culturalLevels)
    {
        // Configure ambient audio sources with environmental levels
        if (ambientSources.Length > 0 && oceanSounds.Length > 0)
```

```
        {
            ambientSources[0].clip = oceanSounds[UnityEngine.Random.Range(0,
oceanSounds.Length)];
            ambientSources[0].volume = oceanLevels[0];
            ambientSources[0].loop = true;
            ambientSources[0].Play();
        }

        if (ambientSources.Length > 1 && windSounds.Length > 0)
        {
            ambientSources[1].clip = windSounds[UnityEngine.Random.Range(0,
windSounds.Length)];
            ambientSources[1].volume = windLevels[0];
            ambientSources[1].loop = true;
            ambientSources[1].Play();
        }

        // Setup cultural audio with respect and authenticity
        SetupCulturalAudio(culturalLevels[0]);
    }

    void SetupCulturalAudio(float intensity)
    {
        // Handle cultural audio with appropriate sensitivity
        if (culturalSources.Length > 0)
        {
            // Boduberu drumming (traditional music)
            if (boduberuDrumming.Length > 0)
            {
```

```
            culturalSources[0].clip =
boduberuDrumming[UnityEngine.Random.Range(0,
boduberuDrumming.Length)];
            culturalSources[0].volume = intensity * 0.7f; // Respectful volume
            culturalSources[0].loop = false;
        }

        // Environmental cultural sounds (fishing, etc.)
        if (fishingSounds.Length > 0)
        {
            culturalSources[1].clip = fishingSounds[UnityEngine.Random.Range(0,
fishingSounds.Length)];
            culturalSources[1].volume = intensity * 0.5f;
            culturalSources[1].loop = true;
            culturalSources[1].Play();
        }
    }
  }

  public void PlayMaldivianEnvironmentalSound(Vector3 position, string
soundType)
  {
    AudioClip clip = GetMaldivianEnvironmentalClip(soundType);
    if (clip != null)
    {
      Play3DSound(clip, position, 1.0f, sfxGroup);
    }
  }

  AudioClip GetMaldivianEnvironmentalClip(string soundType)
  {
```

```
        return soundType switch
        {
            "ocean" => oceanSounds[UnityEngine.Random.Range(0,
oceanSounds.Length)],
            "wind" => windSounds[UnityEngine.Random.Range(0,
windSounds.Length)],
            "wildlife" => wildlifeSounds[UnityEngine.Random.Range(0,
wildlifeSounds.Length)],
            "cultural" => culturalSounds[UnityEngine.Random.Range(0,
culturalSounds.Length)],
            "fishing" => fishingSounds[UnityEngine.Random.Range(0,
fishingSounds.Length)],
            _ => null
        };
    }

    public void Play3DSound(AudioClip clip, Vector3 position, float volume,
AudioMixerGroup mixerGroup)
    {
        // Find available audio source
        AudioSource source = GetAvailableSource(sfxSources);
        if (source != null && clip != null)
        {
            source.transform.position = position;
            source.clip = clip;
            source.volume = volume;
            source.outputAudioMixerGroup = mixerGroup;
            source.Play();
        }
    }
```

```
    AudioSource GetAvailableSource(AudioSource[] sources)
    {
        foreach (AudioSource source in sources)
        {
            if (!source.isPlaying)
            {
                return source;
            }
        }
        return sources[0]; // Fallback to first source
    }

    public void PlayBoduberuRhythm(Vector3 position, float intensity)
    {
        if (boduberuDrumming.Length > 0)
        {
            AudioClip rhythm = boduberuDrumming[UnityEngine.Random.Range(0,
boduberuDrumming.Length)];
            Play3DSound(rhythm, position, intensity * 0.8f, culturalGroup);
        }
    }

    public void SetMasterVolume(float volume)
    {
        if (masterMixer != null)
        {
            masterMixer.SetFloat("MasterVolume", Mathf.Log10(volume) * 20);
        }
    }

    public void SetCulturalAudioVolume(float volume)
```

```
{
    if (masterMixer != null)
    {
        masterMixer.SetFloat("CulturalVolume", Mathf.Log10(volume) * 20);
    }
}

public void UpdateListenerPosition(Vector3 position)
{
    if (audioListener != null)
    {
        audioListener.position = position;
    }
}

void OnDestroy()
{
    // Cleanup audio sources
    foreach (AudioSource source in ambientSources)
    {
        if (source != null) Destroy(source.gameObject);
    }
    foreach (AudioSource source in culturalSources)
    {
        if (source != null) Destroy(source.gameObject);
    }
    foreach (AudioSource source in sfxSources)
    {
        if (source != null) Destroy(source.gameObject);
    }
}
```

```
}
```

28. TimeSystem.cs - Day/Night Cycle Integration

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class TimeSystem : MonoBehaviour
{
    [BurstCompile]
    struct MaldivianTimeCalculation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> gameTimes;
        [WriteOnly] public NativeArray<float> prayerTimes;
        [WriteOnly] public NativeArray<int> timePhases;
        [WriteOnly] public NativeArray<float> lightingIntensity;

        public float latitude;
        public float longitude;
        public int dayOfYear;

        public void Execute(int index)
        {
            float gameTime = gameTimes[index];

            // Calculate prayer times for Maldives
            float prayerTime = CalculatePrayerTime(gameTime);
```

```
    int phase = DetermineTimePhase(gameTime);
    float lighting = CalculateLightingIntensity(gameTime);

    prayerTimes[index] = prayerTime;
    timePhases[index] = phase;
    lightingIntensity[index] = lighting;
}

[BurstCompile]
float CalculatePrayerTime(float time)
{
    // Simplified prayer time calculation for Maldives (4.1755° N, 73.5093° E)
    float baseTime = time;

    // Fajr: ~5:00 AM
    if (time >= 5.0f && time < 5.5f) return 5.0f;
    // Dhuhr: ~12:00 PM
    else if (time >= 12.0f && time < 12.5f) return 12.0f;
    // Asr: ~3:30 PM
    else if (time >= 15.5f && time < 16.0f) return 15.5f;
    // Maghrib: ~6:00 PM
    else if (time >= 18.0f && time < 18.5f) return 18.0f;
    // Isha: ~7:00 PM
    else if (time >= 19.0f && time < 19.5f) return 19.0f;

    return -1.0f; // Not a prayer time
}

[BurstCompile]
int DetermineTimePhase(float time)
{
```

```
        // Maldivian cultural time phases
        if (time >= 5.0f && time < 6.0f) return 1;   // Early Morning (Fajr)
        else if (time >= 6.0f && time < 9.0f) return 2;   // Morning Fishing
        else if (time >= 9.0f && time < 12.0f) return 3;  // Late Morning
        else if (time >= 12.0f && time < 15.0f) return 4; // Midday (Dhuhr)
        else if (time >= 15.0f && time < 18.0f) return 5; // Afternoon (Asr)
        else if (time >= 18.0f && time < 19.0f) return 6; // Evening (Maghrib)
        else if (time >= 19.0f && time < 21.0f) return 7; // Night (Isha)
        else return 8; // Late Night
    }

    [BurstCompile]
    float CalculateLightingIntensity(float time)
    {
        // Calculate sun intensity based on time
        float sunAngle = (time - 12.0f) / 12.0f * math.PI; // -π to π
        float intensity = math.max(0.0f, math.cos(sunAngle));

        // Add some ambient light during night (moon/stars)
        if (intensity < 0.1f)
        {
            intensity = 0.1f + math.sin(time * 0.5f) * 0.05f; // Moonlight
        }

        return intensity;
    }
}

[BurstCompile]
struct CulturalTimeEvents : IJob
{
```

```
public NativeArray<bool> fishingTimeActive;
public NativeArray<bool> prayerTimeActive;
public NativeArray<bool> marketTimeActive;
public NativeArray<bool> socialTimeActive;
public float currentTime;

public void Execute()
{
    CalculateCulturalActivities();
}

[BurstCompile]
void CalculateCulturalActivities()
{
    // Traditional Maldivian daily schedule
    fishingTimeActive[0] = IsFishingTime(currentTime);
    prayerTimeActive[0] = IsPrayerTime(currentTime);
    marketTimeActive[0] = IsMarketTime(currentTime);
    socialTimeActive[0] = IsSocialTime(currentTime);
}

[BurstCompile]
bool IsFishingTime(float time)
{
    // Traditional fishing times: early morning and late afternoon
    return (time >= 5.5f && time < 8.0f) || (time >= 16.0f && time < 18.0f);
}

[BurstCompile]
bool IsPrayerTime(float time)
{
```

```
        // Check if within 15 minutes of any prayer time
        float[] prayerTimes = { 5.25f, 12.0f, 15.75f, 18.0f, 19.0f };
        foreach (float prayerTime in prayerTimes)
        {
            if (math.abs(time - prayerTime) < 0.25f) return true;
        }
        return false;
    }

    [BurstCompile]
    bool IsMarketTime(float time)
    {
        // Market active during morning and evening
        return (time >= 8.0f && time < 11.0f) || (time >= 17.0f && time < 20.0f);
    }

    [BurstCompile]
    bool IsSocialTime(float time)
    {
        // Evening social gatherings
        return time >= 19.0f && time < 22.0f;
    }
}

public static TimeSystem Instance { get; private set; }

[Header("Time Settings")]
public float timeScale = 60f; // 1 real minute = 1 game hour
public float currentTime = 12.0f; // Start at noon
public int currentDay = 1;
```

```
    [Header("Maldivian Coordinates")]
    public float latitude = 4.1755f; // Malé latitude
    public float longitude = 73.5093f; // Malé longitude

    [Header("Time Events")]
    public UnityEngine.Events.UnityAction<float> onTimeChanged;
    public UnityEngine.Events.UnityAction<float> onPrayerTime;
    public UnityEngine.Events.UnityAction<int> onTimePhaseChanged;
    public UnityEngine.Events.UnityAction onNewDay;

    // Cultural time states
    public bool isFishingTime { get; private set; }
    public bool isPrayerTime { get; private set; }
    public bool isMarketTime { get; private set; }
    public bool isSocialTime { get; private set; }

    private int currentTimePhase;
    private float lastPrayerTime = -1f;

    void Awake()
    {
        Instance = this;
        InitializeTimeSystem();
    }

    void InitializeTimeSystem()
    {
        int timeCalculations = 24; // One for each hour
        var gameTimes = new NativeArray<float>(timeCalculations,
Allocator.TempJob);
```

```
        var prayerTimes = new NativeArray<float>(timeCalculations,
Allocator.TempJob);
        var timePhases = new NativeArray<int>(timeCalculations,
Allocator.TempJob);
        var lightingIntensity = new NativeArray<float>(timeCalculations,
Allocator.TempJob);

        var fishingTime = new NativeArray<bool>(1, Allocator.TempJob);
        var prayerTime = new NativeArray<bool>(1, Allocator.TempJob);
        var marketTime = new NativeArray<bool>(1, Allocator.TempJob);
        var socialTime = new NativeArray<bool>(1, Allocator.TempJob);

        // Initialize time data
        for (int i = 0; i < timeCalculations; i++)
        {
            gameTimes[i] = (float)i;
        }

        var maldivianTimeJob = new MaldivianTimeCalculation
        {
            gameTimes = gameTimes,
            prayerTimes = prayerTimes,
            timePhases = timePhases,
            lightingIntensity = lightingIntensity,
            latitude = latitude,
            longitude = longitude,
            dayOfYear = System.DateTime.Now.DayOfYear
        };

        var culturalJob = new CulturalTimeEvents
        {
```

```
            fishingTimeActive = fishingTime,
            prayerTimeActive = prayerTime,
            marketTimeActive = marketTime,
            socialTimeActive = socialTime,
            currentTime = currentTime
        };

        JobHandle timeHandle = maldivianTimeJob.Schedule(timeCalculations, 4);
        JobHandle culturalHandle = culturalJob.Schedule(timeHandle);
        culturalHandle.Complete();

        // Update cultural states
        isFishingTime = fishingTime[0];
        isPrayerTime = prayerTime[0];
        isMarketTime = marketTime[0];
        isSocialTime = socialTime[0];

        gameTimes.Dispose();
        prayerTimes.Dispose();
        timePhases.Dispose();
        lightingIntensity.Dispose();
        fishingTime.Dispose();
        prayerTime.Dispose();
        marketTime.Dispose();
        socialTime.Dispose();
    }

    void Update()
    {
        // Update game time
        float previousTime = currentTime;
```

```
currentTime += (Time.deltaTime * timeScale) / 3600f; // Convert to hours

// Handle day rollover
if (currentTime >= 24f)
{
    currentTime -= 24f;
    currentDay++;
    onNewDay?.Invoke();
}

// Check for time changes
if (math.floor(currentTime) != math.floor(previousTime))
{
    onTimeChanged?.Invoke(currentTime);
    UpdateCulturalActivities();
}

// Check for prayer times
float currentPrayerTime = GetCurrentPrayerTime();
if (currentPrayerTime != lastPrayerTime && currentPrayerTime > 0)
{
    lastPrayerTime = currentPrayerTime;
    onPrayerTime?.Invoke(currentPrayerTime);
}

// Update time phase
int newPhase = GetCurrentTimePhase();
if (newPhase != currentTimePhase)
{
    currentTimePhase = newPhase;
    onTimePhaseChanged?.Invoke(newPhase);
```

```
    }
}

void UpdateCulturalActivities()
{
    int culturalChecks = 4;
    var fishingTime = new NativeArray<bool>(1, Allocator.TempJob);
    var prayerTime = new NativeArray<bool>(1, Allocator.TempJob);
    var marketTime = new NativeArray<bool>(1, Allocator.TempJob);
    var socialTime = new NativeArray<bool>(1, Allocator.TempJob);

    var culturalJob = new CulturalTimeEvents
    {
        fishingTimeActive = fishingTime,
        prayerTimeActive = prayerTime,
        marketTimeActive = marketTime,
        socialTimeActive = socialTime,
        currentTime = currentTime
    };

    JobHandle handle = culturalJob.Schedule();
    handle.Complete();

    isFishingTime = fishingTime[0];
    isPrayerTime = prayerTime[0];
    isMarketTime = marketTime[0];
    isSocialTime = socialTime[0];

    fishingTime.Dispose();
    prayerTime.Dispose();
    marketTime.Dispose();
```

```
    socialTime.Dispose();
}

float GetCurrentPrayerTime()
{
    // Check if current time matches any prayer time
    float[] prayerTimes = { 5.25f, 12.0f, 15.75f, 18.0f, 19.0f };
    foreach (float prayerTime in prayerTimes)
    {
        if (math.abs(currentTime - prayerTime) < 0.01f)
        {
            return prayerTime;
        }
    }
    return -1f;
}

int GetCurrentTimePhase()
{
    // Determine current cultural time phase
    if (currentTime >= 5.0f && currentTime < 6.0f) return 1;
    else if (currentTime >= 6.0f && currentTime < 9.0f) return 2;
    else if (currentTime >= 9.0f && currentTime < 12.0f) return 3;
    else if (currentTime >= 12.0f && currentTime < 15.0f) return 4;
    else if (currentTime >= 15.0f && currentTime < 18.0f) return 5;
    else if (currentTime >= 18.0f && currentTime < 19.0f) return 6;
    else if (currentTime >= 19.0f && currentTime < 21.0f) return 7;
    else return 8;
}

public float GetSunIntensity()
```

```
{
    // Calculate sun intensity for lighting
    float sunAngle = (currentTime - 12.0f) / 12.0f * Mathf.PI;
    return Mathf.Max(0.1f, Mathf.Cos(sunAngle));
}

public string GetTimeOfDayString()
{
    int hours = Mathf.FloorToInt(currentTime);
    int minutes = Mathf.FloorToInt((currentTime - hours) * 60);
    return $"{hours:00}:{minutes:00}";
}

public string GetDhivehiTimeString()
{
    // Return time in Dhivehi format
    string timeStr = GetTimeOfDayString();
    return $"fiÜfi¶fiÉfi™fiàfi¶fiÉfi™ {timeStr}"; // "Time: HH:MM"
}

public bool IsNightTime()
{
    return currentTime < 6.0f || currentTime > 18.0f;
}

public bool IsDayTime()
{
    return currentTime >= 6.0f && currentTime <= 18.0f;
}

public float GetTimeUntilNextPrayer()
```

```
    {
        float[] prayerTimes = { 5.25f, 12.0f, 15.75f, 18.0f, 19.0f };
        foreach (float prayerTime in prayerTimes)
        {
            if (currentTime < prayerTime)
            {
                return prayerTime - currentTime;
            }
        }

        // Next prayer is tomorrow's Fajr
        return (24.0f - currentTime) + 5.25f;
    }

    public void SetTimeScale(float scale)
    {
        timeScale = math.max(1f, scale); // Minimum 1:1 ratio
    }

    public void SkipToTime(float targetTime)
    {
        currentTime = math.saturate(targetTime) * 24f;
        UpdateCulturalActivities();
    }
}
```

29. LightingSystem.cs - Dynamic Lighting

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
```

```
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class LightingSystem : MonoBehaviour
{
    [BurstCompile]
    struct MaldivianSunlightSimulation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> timeValues;
        [WriteOnly] public NativeArray<float3> sunDirections;
        [WriteOnly] public NativeArray<float3> sunColors;
        [WriteOnly] public NativeArray<float> sunIntensities;

        public float latitude;
        public float longitude;
        public int dayOfYear;

        public void Execute(int index)
        {
            float time = timeValues[index];

            // Calculate Maldivian sunlight parameters
            float3 direction = CalculateSunDirection(time);
            float3 color = CalculateSunColor(time);
            float intensity = CalculateSunIntensity(time);

            sunDirections[index] = direction;
            sunColors[index] = color;
            sunIntensities[index] = intensity;
        }
```

```
    [BurstCompile]
    float3 CalculateSunDirection(float time)
    {
        // Simplified sun position for Maldives (near equator)
        float hourAngle = (time - 12.0f) * 15.0f; // 15 degrees per hour
        float declination = 0.0f; // Near equator, minimal seasonal variation

        float radHour = math.radians(hourAngle);
        float radDecl = math.radians(declination);
        float radLat = math.radians(latitude);

        float elevation = math.asin(math.sin(radLat) * math.sin(radDecl) +
                                    math.cos(radLat) * math.cos(radDecl) *
math.cos(radHour));

        float azimuth = math.atan2(math.sin(radHour),
                                   math.cos(radHour) * math.sin(radLat) - math.tan(radDecl)
* math.cos(radLat));

        // Convert to direction vector
        float3 direction = new float3(
            math.cos(elevation) * math.sin(azimuth),
            math.sin(elevation),
            math.cos(elevation) * math.cos(azimuth)
        );

        return direction;
    }

    [BurstCompile]
```

```
float3 CalculateSunColor(float time)
{
    // Tropical sunlight color variations
    if (time < 6.0f || time > 18.0f) // Night
    {
        return new float3(0.1f, 0.1f, 0.2f); // Moonlight blue
    }
    else if (time < 8.0f || time > 16.0f) // Dawn/Dusk
    {
        return new float3(1.0f, 0.6f, 0.3f); // Tropical sunrise/sunset
    }
    else if (time < 10.0f || time > 14.0f) // Morning/Afternoon
    {
        return new float3(1.0f, 0.9f, 0.7f); // Warm tropical light
    }
    else // Midday
    {
        return new float3(1.0f, 1.0f, 0.9f); // Bright tropical sun
    }
}

[BurstCompile]
float CalculateSunIntensity(float time)
{
    // Tropical sun intensity (very bright at peak)
    if (time < 5.5f || time > 18.5f) return 0.0f; // Night

    float normalizedTime = (time - 5.5f) / 13.0f; // 0 to 1 over daylight hours
    float intensity = math.sin(normalizedTime * math.PI);

    return intensity * 1.2f; // Brighter than temperate regions
```

```
    }
}

[BurstCompile]
struct CulturalLightingCalculation : IJob
{
    public NativeArray<float3> artificialLightColors;
    public NativeArray<float> artificialLightIntensities;
    public float currentTime;
    public bool isPrayerTime;
    public bool isCulturalEvent;

    public void Execute()
    {
        CalculateCulturalLighting();
    }

    [BurstCompile]
    void CalculateCulturalLighting()
    {
        for (int i = 0; i < artificialLightColors.Length; i++)
        {
            // Respectful artificial lighting for Maldivian culture
            float3 baseColor = new float3(1.0f, 0.9f, 0.7f); // Warm white
            float intensity = 0.8f;

            // Dim during prayer times
            if (isPrayerTime)
            {
                intensity *= 0.5f;
                baseColor = new float3(0.8f, 0.6f, 0.4f); // Softer, warmer
```

```
            }

            // Enhanced during cultural events
            if (isCulturalEvent)
            {
                intensity *= 1.2f;
                baseColor = new float3(1.0f, 0.8f, 0.5f); // Golden cultural lighting
            }

            // Evening/night adjustments
            if (currentTime > 18.0f || currentTime < 6.0f)
            {
                intensity *= 0.7f; // Respectful night lighting
            }

            artificialLightColors[i] = baseColor;
            artificialLightIntensities[i] = intensity;
        }
    }
}

public static LightingSystem Instance { get; private set; }

[Header("Lighting Components")]
public Light sunLight;
public Light moonLight;
public Light[] artificialLights;

[Header("Maldivian Lighting Settings")]
public Gradient maldivianSunGradient;
public Gradient maldivianSkyGradient;
```

```
    public AnimationCurve sunIntensityCurve;
    public float equatorialSunIntensity = 1.2f;

    [Header("Cultural Lighting")]
    public bool respectPrayerTimes = true;
    public bool dimLightsDuringPrayer = true;
    public float prayerTimeDimming = 0.5f;

    private float currentSunIntensity;
    private Color currentSunColor;
    private Vector3 currentSunDirection;
    private bool isPrayerTimeActive;

    void Awake()
    {
        Instance = this;
        InitializeLightingSystem();
    }

    void InitializeLightingSystem()
    {
        // Setup lighting components
        SetupSunLight();
        SetupMoonLight();
        SetupArtificialLights();

        int timeCalculations = 24; // Hourly calculations
        var timeValues = new NativeArray<float>(timeCalculations,
Allocator.TempJob);
        var sunDirections = new NativeArray<float3>(timeCalculations,
Allocator.TempJob);
```

```
        var sunColors = new NativeArray<float3>(timeCalculations,
Allocator.TempJob);
        var sunIntensities = new NativeArray<float>(timeCalculations,
Allocator.TempJob);

        var artificialColors = new NativeArray<float3>(artificialLights.Length,
Allocator.TempJob);
        var artificialIntensities = new NativeArray<float>(artificialLights.Length,
Allocator.TempJob);

        // Initialize time values
        for (int i = 0; i < timeCalculations; i++)
        {
            timeValues[i] = (float)i;
        }

        var sunlightJob = new MaldivianSunlightSimulation
        {
            timeValues = timeValues,
            sunDirections = sunDirections,
            sunColors = sunColors,
            sunIntensities = sunIntensities,
            latitude = 4.1755f, // Malé latitude
            longitude = 73.5093f, // Malé longitude
            dayOfYear = System.DateTime.Now.DayOfYear
        };

        var culturalJob = new CulturalLightingCalculation
        {
            artificialLightColors = artificialColors,
            artificialLightIntensities = artificialIntensities,
```

```
            currentTime = TimeSystem.Instance?.currentTime ?? 12.0f,
            isPrayerTime = TimeSystem.Instance?.isPrayerTime ?? false,
            isCulturalEvent = false
        };

        JobHandle sunlightHandle = sunlightJob.Schedule(timeCalculations, 4);
        JobHandle culturalHandle = culturalJob.Schedule(sunlightHandle);
        culturalHandle.Complete();

        // Store lighting data for later use
        CacheLightingData(sunDirections, sunColors, sunIntensities);

        // Apply cultural lighting settings
        ApplyCulturalLighting(artificialColors, artificialIntensities);

        timeValues.Dispose();
        sunDirections.Dispose();
        sunColors.Dispose();
        sunIntensities.Dispose();
        artificialColors.Dispose();
        artificialIntensities.Dispose();
    }

    void SetupSunLight()
    {
        if (sunLight == null)
        {
            GameObject sunObj = GameObject.Find("Sun") ?? new
GameObject("Sun");
            sunLight = sunObj.GetComponent<Light>();
            if (sunLight == null)
```

```
            {
                sunLight = sunObj.AddComponent<Light>();
                sunLight.type = LightType.Directional;
            }
        }

        // Configure sun for tropical lighting
        sunLight.color = Color.white;
        sunLight.intensity = equatorialSunIntensity;
        sunLight.shadows = LightShadows.Soft;
        sunLight.shadowStrength = 0.8f;
        sunLight.shadowResolution = LightShadowResolution.High;
    }

    void SetupMoonLight()
    {
        if (moonLight == null)
        {
            GameObject moonObj = GameObject.Find("Moon") ?? new
GameObject("Moon");
            moonLight = moonObj.GetComponent<Light>();
            if (moonLight == null)
            {
                moonLight = moonObj.AddComponent<Light>();
                moonLight.type = LightType.Directional;
            }
        }

        moonLight.color = new Color(0.8f, 0.8f, 1.0f);
        moonLight.intensity = 0.3f;
        moonLight.shadows = LightShadows.Hard;
```

```
    }

    void SetupArtificialLights()
    {
        if (artificialLights == null || artificialLights.Length == 0)
        {
            // Find existing artificial lights
            artificialLights = FindObjectsOfType<Light>();
            System.Collections.Generic.List<Light> artificialList = new
System.Collections.Generic.List<Light>();

            foreach (Light light in artificialLights)
            {
                if (light.type != LightType.Directional)
                {
                    artificialList.Add(light);
                }
            }

            artificialLights = artificialList.ToArray();
        }

        // Configure artificial lights for cultural sensitivity
        foreach (Light light in artificialLights)
        {
            light.color = new Color(1.0f, 0.9f, 0.7f); // Warm white
            light.intensity = 0.8f;
            light.range = 10f;
            light.shadows = LightShadows.Soft;
        }
    }
```

```
    void CacheLightingData(NativeArray<float3> directions, NativeArray<float3>
colors, NativeArray<float> intensities)
    {
        // Store lighting data for runtime use
        // This could be optimized with lookup tables
    }

    void ApplyCulturalLighting(NativeArray<float3> colors, NativeArray<float>
intensities)
    {
        // Apply cultural lighting settings to artificial lights
        for (int i = 0; i < math.min(artificialLights.Length, colors.Length); i++)
        {
            Color lightColor = new Color(colors[i].x, colors[i].y, colors[i].z);
            artificialLights[i].color = lightColor;
            artificialLights[i].intensity = intensities[i];
        }
    }

    void Update()
    {
        // Update lighting based on current time
        float currentTime = TimeSystem.Instance?.currentTime ?? 12.0f;
        UpdateSunlight(currentTime);
        UpdateMoonlight(currentTime);
        UpdateArtificialLights(currentTime);

        // Handle prayer time lighting adjustments
        bool prayerTime = TimeSystem.Instance?.isPrayerTime ?? false;
        if (prayerTime != isPrayerTimeActive)
```

```
        {
            isPrayerTimeActive = prayerTime;
            ApplyPrayerTimeLighting(prayerTime);
        }
    }

    void UpdateSunlight(float time)
    {
        if (sunLight == null) return;

        // Calculate sun position and intensity
        float sunAngle = (time - 12.0f) / 12.0f * Mathf.PI;
        float intensity = Mathf.Max(0.0f, Mathf.Cos(sunAngle)) *
equatorialSunIntensity;

        // Update sun rotation
        float elevation = intensity * 90f; // Simple elevation calculation
        float azimuth = (time / 24f) * 360f; // Full rotation in 24 hours

        sunLight.transform.rotation = Quaternion.Euler(elevation, azimuth, 0);

        // Update sun intensity and color
        sunLight.intensity = intensity;
        sunLight.color = GetSunColorForTime(time);

        // Enable/disable sun based on time
        sunLight.enabled = intensity > 0.01f;
    }

    void UpdateMoonlight(float time)
    {
```

```
        if (moonLight == null) return;

        // Moon is active during night
        bool isNight = time < 6.0f || time > 18.0f;
        moonLight.enabled = isNight;

        if (isNight)
        {
            // Calculate moon phase and intensity
            float moonPhase = CalculateMoonPhase();
            moonLight.intensity = 0.2f + moonPhase * 0.3f;

            // Moon position (opposite sun)
            moonLight.transform.rotation = Quaternion.Euler(45f, (time / 24f) * 360f +
180f, 0);
        }
    }

    void UpdateArtificialLights(float time)
    {
        bool isNight = time < 6.0f || time > 18.0f;

        foreach (Light light in artificialLights)
        {
            if (light != null)
            {
                // Enable artificial lights during night
                light.enabled = isNight;

                if (isNight)
                {
```

```
            // Adjust intensity based on time and cultural factors
            float baseIntensity = 0.8f;
            if (time > 22.0f || time < 5.0f)
            {
                baseIntensity = 0.4f; // Dim late night/early morning
            }

            light.intensity = baseIntensity;
        }
    }
  }
}

Color GetSunColorForTime(float time)
{
  if (time < 6.0f || time > 18.0f) return Color.black;
  else if (time < 8.0f || time > 16.0f) return new Color(1.0f, 0.7f, 0.4f);
  else if (time < 10.0f || time > 14.0f) return new Color(1.0f, 0.9f, 0.7f);
  else return Color.white;
}

float CalculateMoonPhase()
{
  // Simple moon phase calculation
  float lunarCycle = 29.53f; // days
  float daysSinceNew = (Time.time / 86400f) % lunarCycle;
  return daysSinceNew / lunarCycle;
}

void ApplyPrayerTimeLighting(bool isPrayerTime)
{
```

```
    if (!respectPrayerTimes) return;

    foreach (Light light in artificialLights)
    {
        if (light != null)
        {
            if (isPrayerTime && dimLightsDuringPrayer)
            {
                light.intensity *= prayerTimeDimming;
                light.color = new Color(0.9f, 0.7f, 0.5f); // Warmer, softer
            }
            else
            {
                // Restore normal lighting
                light.intensity = 0.8f;
                light.color = new Color(1.0f, 0.9f, 0.7f);
            }
        }
    }
}

public float GetCurrentSunIntensity()
{
    return currentSunIntensity;
}

public Color GetCurrentSunColor()
{
    return currentSunColor;
}
```

```
    public Vector3 GetCurrentSunDirection()
    {
        return currentSunDirection;
    }

    public void SetCulturalEventLighting(bool active)
    {
        // Apply special lighting for cultural events
        foreach (Light light in artificialLights)
        {
            if (light != null)
            {
                if (active)
                {
                    light.color = new Color(1.0f, 0.8f, 0.5f); // Golden cultural lighting
                    light.intensity = 1.0f;
                }
                else
                {
                    light.color = new Color(1.0f, 0.9f, 0.7f); // Normal warm white
                    light.intensity = 0.8f;
                }
            }
        }
    }
}
```

30. NetworkingSystem.cs - Multiplayer Framework

```
using UnityEngine;
using Unity.Burst;
```

```
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using Unity.Netcode;
using Unity.Networking.Transport;

[BurstCompile]
public class NetworkingSystem : MonoBehaviour
{
    [BurstCompile]
    struct NetworkDataCompression : IJobParallelFor
    {
        [ReadOnly] public NativeArray<byte> uncompressedData;
        [WriteOnly] public NativeArray<byte> compressedData;

        public void Execute(int index)
        {
            // Simple RLE compression for network data
            compressedData[index] = (byte)(uncompressedData[index] ^ 0xAA);
        }
    }

    [BurstCompile]
    struct CulturalDataValidation : IJob
    {
        public NativeArray<byte> culturalData;
        public NativeArray<bool> validationResults;

        public void Execute()
        {
            ValidateCulturalIntegrity();
```

```
    }

    [BurstCompile]
    void ValidateCulturalIntegrity()
    {
        // Ensure cultural data integrity during network sync
        bool isValid = true;

        // Basic validation: check for data corruption
        for (int i = 0; i < culturalData.Length; i++)
        {
            if (culturalData[i] == 0xFF) // Invalid byte pattern
            {
                isValid = false;
                break;
            }
        }

        validationResults[0] = isValid;
    }
}

public static NetworkingSystem Instance { get; private set; }

[Header("Network Configuration")]
public string serverURL = "https://rvac-server.maldives.game";
public int maxPlayers = 20;
public int networkUpdateRate = 30; // Hz
public float networkTimeout = 10f;

[Header("Cultural Network Features")]
```

```
public bool enableCulturalDataSync = true;
public bool respectLocalCulturalSettings = true;
public bool enablePrayerTimeSync = true;

[Header("Cloud Services")]
public bool enableCloudSaves = true;
public bool enableCrossPlatformSync = true;
public int maxRetries = 3;

private NetworkDriver networkDriver;
private NetworkConnection connection;
private bool isConnected;
private bool isHost;
private int localPlayerID;

// Cultural network data
private MaldivianCulturalData localCulturalData;
private MaldivianCulturalData[] remoteCulturalData;

[System.Serializable]
public class MaldivianCulturalData
{
    public int playerID;
    public string playerName;
    public string dhivehiPlayerName;
    public float respectLevel;
    public bool hasCompletedPrayer;
    public bool hasParticipatedInCulturalEvent;
    public int islandsVisited;
    public string[] discoveredCulturalSites;
    public float culturalKnowledgeScore;
```

```
    public bool respectsLocalCustoms;
}

void Awake()
{
    Instance = this;
    InitializeNetworkingSystem();
}

void InitializeNetworkingSystem()
{
    // Initialize network driver
    NetworkSettings settings = new NetworkSettings();
    settings.WithNetworkConfigParameters(
        maxFrameTime: 0, // No limit
        disconnectTimeoutMS: (int)(networkTimeout * 1000)
    );

    networkDriver = NetworkDriver.Create(settings);

    // Initialize cultural data
    localCulturalData = new MaldivianCulturalData
    {
        playerID = 0,
        playerName = "Player",
        dhivehiPlayerName = "fiÜfi™fiÖfi®fiàfi®fiÉfi®fiáfi¨fiáfi∞",
        respectLevel = 0.5f,
        hasCompletedPrayer = false,
        hasParticipatedInCulturalEvent = false,
        islandsVisited = 0,
        discoveredCulturalSites = new string[0],
```

```
            culturalKnowledgeScore = 0.0f,
            respectsLocalCustoms = true
        };

        remoteCulturalData = new MaldivianCulturalData[maxPlayers];

        int networkDataSize = 1000;
        var uncompressedData = new NativeArray<byte>(networkDataSize,
Allocator.TempJob);
        var compressedData = new NativeArray<byte>(networkDataSize,
Allocator.TempJob);
        var culturalData = new NativeArray<byte>(networkDataSize,
Allocator.TempJob);
        var validationResults = new NativeArray<bool>(1, Allocator.TempJob);

        // Initialize network data
        for (int i = 0; i < networkDataSize; i++)
        {
            uncompressedData[i] = (byte)(i % 256);
            culturalData[i] = (byte)(UnityEngine.Random.Range(0, 255));
        }

        var compressionJob = new NetworkDataCompression
        {
            uncompressedData = uncompressedData,
            compressedData = compressedData
        };

        var validationJob = new CulturalDataValidation
        {
            culturalData = culturalData,
```

```
            validationResults = validationResults
        };

        JobHandle compressionHandle =
compressionJob.Schedule(networkDataSize, 64);
        JobHandle validationHandle = validationJob.Schedule(compressionHandle);
        validationHandle.Complete();

        bool isDataValid = validationResults[0];
        Debug.Log($"Cultural data validation: {isDataValid}");

        uncompressedData.Dispose();
        compressedData.Dispose();
        culturalData.Dispose();
        validationResults.Dispose();

        StartNetworkServices();
    }

    void StartNetworkServices()
    {
        // Start connection to server
        StartCoroutine(ConnectToServer());

        // Initialize cloud save sync
        if (enableCloudSaves)
        {
            InitializeCloudSaveSync();
        }

        // Start cultural data synchronization
```

```
    if (enableCulturalDataSync)
    {
        StartCulturalDataSync();
    }

    // Start network update loop
    InvokeRepeating("NetworkUpdate", 0f, 1f / networkUpdateRate);
}

System.Collections.IEnumerator ConnectToServer()
{
    // Simulate network connection
    yield return new WaitForSeconds(1f);

    // Connect to Maldivian game server
    NetworkEndPoint endpoint = NetworkEndPoint.Parse(serverURL, 7777);

    if (networkDriver.Bind(endpoint) != 0)
    {
        Debug.LogError("Failed to bind to network endpoint");
        yield break;
    }

    if (networkDriver.Listen() != 0)
    {
        Debug.LogError("Failed to listen on network endpoint");
        yield break;
    }

    isConnected = true;
    Debug.Log("Connected to Maldivian game server");
```

```
    }

    void InitializeCloudSaveSync()
    {
        // Setup cloud save synchronization
        InvokeRepeating("SyncCloudSaves", 30f, 300f); // Every 5 minutes
    }

    void StartCulturalDataSync()
    {
        // Start syncing cultural data with other players
        InvokeRepeating("SyncCulturalData", 10f, 60f); // Every minute
    }

    void NetworkUpdate()
    {
        if (!isConnected) return;

        // Update network driver
        networkDriver.ScheduleUpdate().Complete();

        // Handle incoming connections
        NetworkConnection incomingConnection;
        while ((incomingConnection = networkDriver.Accept()) !=
default(NetworkConnection))
        {
            HandleNewConnection(incomingConnection);
        }

        // Handle existing connection
        if (connection != default(NetworkConnection) && connection.IsCreated)
```

```
        {
            HandleConnectionData();
        }
    }

    void HandleNewConnection(NetworkConnection newConnection)
    {
        // Handle new player connection with cultural sensitivity
        int playerID = GetNextPlayerID();

        // Send cultural welcome message
        SendCulturalWelcomeMessage(playerID);

        // Request cultural data from new player
        RequestCulturalData(playerID);

        Debug.Log($"New player connected with ID: {playerID}");
    }

    void HandleConnectionData()
    {
        NetworkEvent.Type eventType;
        while ((eventType = connection.PopEvent(networkDriver, out
DataStreamReader stream)) != NetworkEvent.Type.Empty)
        {
            switch (eventType)
            {
                case NetworkEvent.Type.Data:
                    ProcessNetworkData(stream);
                    break;
                case NetworkEvent.Type.Disconnect:
```

```
            HandleDisconnection();
            break;
        }
    }
}

void ProcessNetworkData(DataStreamReader stream)
{
    int dataSize = stream.Length;
    var networkData = new NativeArray<byte>(dataSize, Allocator.Temp);

    // Read data from stream
    for (int i = 0; i < dataSize; i++)
    {
        networkData[i] = stream.ReadByte();
    }

    // Process cultural network data
    ProcessCulturalNetworkData(networkData);

    networkData.Dispose();
}

void ProcessCulturalNetworkData(NativeArray<byte> data)
{
    // Validate cultural data integrity
    var validationData = new NativeArray<byte>(data.Length, Allocator.Temp);
    var validationResults = new NativeArray<bool>(1, Allocator.Temp);

    for (int i = 0; i < data.Length; i++)
    {
```

```
        validationData[i] = data[i];
    }

    var validationJob = new CulturalDataValidation
    {
        culturalData = validationData,
        validationResults = validationResults
    };

    JobHandle handle = validationJob.Schedule();
    handle.Complete();

    bool isValid = validationResults[0];

    if (isValid && respectLocalCulturalSettings)
    {
        // Apply cultural data respecting local settings
        ApplyCulturalNetworkData(data);
    }

    validationData.Dispose();
    validationResults.Dispose();
}

void ApplyCulturalNetworkData(NativeArray<byte> data)
{
    // Deserialize and apply cultural data
    // Implementation depends on specific data format
    Debug.Log("Applied cultural network data");
}
```

```
    void HandleDisconnection()
    {
        isConnected = false;
        connection = default(NetworkConnection);
        Debug.Log("Disconnected from server");
    }

    void SyncCloudSaves()
    {
        if (!enableCloudSaves) return;

        // Upload local save data to cloud
        StartCoroutine(UploadSaveData());

        // Download remote save data
        StartCoroutine(DownloadSaveData());
    }

    System.Collections.IEnumerator UploadSaveData()
    {
        // Simulate cloud upload
        yield return new WaitForSeconds(2f);

        SaveSystem saveSystem = SaveSystem.Instance;
        if (saveSystem != null)
        {
            string saveData = JsonUtility.ToJson(saveSystem.currentSave);

            // Compress and upload save data
            byte[] compressedData =
CompressData(System.Text.Encoding.UTF8.GetBytes(saveData));
```

```
            // Upload to Maldivian cloud server
            Debug.Log($"Uploaded {compressedData.Length} bytes of save data to
cloud");
        }
    }

    System.Collections.IEnumerator DownloadSaveData()
    {
        // Simulate cloud download
        yield return new WaitForSeconds(2f);

        // Download from cloud server
        byte[] downloadedData = new byte[100]; // Simulated downloaded data

        if (downloadedData.Length > 0)
        {
            string saveData =
System.Text.Encoding.UTF8.GetString(DecompressData(downloadedData));

            // Apply downloaded save data
            SaveSystem saveSystem = SaveSystem.Instance;
            if (saveSystem != null)
            {
                saveSystem.currentSave =
JsonUtility.FromJson<SaveSystem.GameSaveData>(saveData);
                Debug.Log("Downloaded and applied cloud save data");
            }
        }
    }
```

```
    void SyncCulturalData()
    {
        if (!enableCulturalDataSync) return;

        // Share local cultural data with network
        ShareLocalCulturalData();

        // Receive cultural data from other players
        ReceiveRemoteCulturalData();
    }

    void ShareLocalCulturalData()
    {
        // Serialize local cultural data
        string culturalDataJson = JsonUtility.ToJson(localCulturalData);
        byte[] culturalDataBytes =
System.Text.Encoding.UTF8.GetBytes(culturalDataJson);

        // Compress and send
        byte[] compressedData = CompressData(culturalDataBytes);

        // Send to network
        if (connection.IsCreated)
        {
            SendNetworkData(compressedData);
        }
    }

    void ReceiveRemoteCulturalData()
    {
        // Process received cultural data from other players
```

```
        for (int i = 0; i < remoteCulturalData.Length; i++)
        {
            if (remoteCulturalData[i] != null && remoteCulturalData[i].playerID != 0)
            {
                // Update UI or game state based on remote cultural data
                UpdateCulturalUI(remoteCulturalData[i]);
            }
        }
    }

    void SendNetworkData(byte[] data)
    {
        if (!connection.IsCreated) return;

        var writer = networkDriver.BeginSend(connection);
        if (writer.IsCreated)
        {
            writer.WriteBytes(data);
            networkDriver.EndSend(writer);
        }
    }

    void SendCulturalWelcomeMessage(int playerID)
    {
        string welcomeMessage = $"Welcome to RAAJJE VAGU AUTO! Respect
Maldivian culture and traditions.";
        byte[] messageBytes =
System.Text.Encoding.UTF8.GetBytes(welcomeMessage);
        SendNetworkData(messageBytes);
    }
```

```
    void RequestCulturalData(int playerID)
    {
        string request = $"REQUEST_CULTURAL_DATA:{playerID}";
        byte[] requestBytes = System.Text.Encoding.UTF8.GetBytes(request);
        SendNetworkData(requestBytes);
    }

    void UpdateCulturalUI(MaldivianCulturalData culturalData)
    {
        // Update UI to show remote player's cultural status
        Debug.Log($"Player {culturalData.playerName} - Cultural Score:
{culturalData.culturalKnowledgeScore}");
    }

    byte[] CompressData(byte[] data)
    {
        // Simple compression for network transmission
        byte[] compressed = new byte[data.Length];
        for (int i = 0; i < data.Length; i++)
        {
            compressed[i] = (byte)(data[i] ^ 0xAA);
        }
        return compressed;
    }

    byte[] DecompressData(byte[] compressedData)
    {
        // Decompress network data
        byte[] decompressed = new byte[compressedData.Length];
        for (int i = 0; i < compressedData.Length; i++)
        {
```

```
            decompressed[i] = (byte)(compressedData[i] ^ 0xAA);
        }
        return decompressed;
    }

    int GetNextPlayerID()
    {
        return ++localPlayerID;
    }

    public void UpdateLocalCulturalData(System.Action<MaldivianCulturalData>
updateAction)
    {
        updateAction?.Invoke(localCulturalData);
        ShareLocalCulturalData(); // Immediately sync changes
    }

    public MaldivianCulturalData GetLocalCulturalData()
    {
        return localCulturalData;
    }

    public bool IsConnected()
    {
        return isConnected;
    }

    public bool IsHost()
    {
        return isHost;
    }
```

```
    void OnDestroy()
    {
        // Cleanup network resources
        if (networkDriver.IsCreated)
        {
            networkDriver.Dispose();
        }

        CancelInvoke();
    }
}
```

---

31. AnalyticsSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System;
using System.Text;

[BurstCompile]
public class AnalyticsSystem : MonoBehaviour
{
    [BurstCompile]
    struct PrivacyCompliantDataCollection : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> gameplayMetrics;
```

```
    [WriteOnly] public NativeArray<float> anonymizedData;
    [WriteOnly] public NativeArray<bool> privacyFlags;

    public float anonymizationThreshold;
    public bool collectPersonalData;

    public void Execute(int index)
    {
        float rawData = gameplayMetrics[index];

        // Apply privacy-compliant anonymization
        float anonymized = AnonymizeData(rawData);
        bool isPrivate = IsSensitiveData(rawData);

        anonymizedData[index] = anonymized;
        privacyFlags[index] = isPrivate;
    }

    [BurstCompile]
    float AnonymizeData(float data)
    {
        // GDPR-compliant data anonymization
        float noise =
Unity.Mathematics.Random.CreateFromIndex((uint)index).NextFloat(-0.1f, 0.1f);
        return data + noise;
    }

    [BurstCompile]
    bool IsSensitiveData(float data)
    {
        // Identify potentially sensitive data points
```

```
        return math.abs(data) > anonymizationThreshold;
    }
}

[BurstCompile]
struct CulturalMetricsAggregation : IJob
{
    public NativeArray<int> culturalInteractions;
    public NativeArray<float> respectScores;
    public NativeArray<int> prayerParticipation;
    public NativeArray<int> traditionalActivities;

    public NativeArray<float> aggregatedMetrics;

    public void Execute()
    {
        AggregateCulturalData();
    }

    [BurstCompile]
    void AggregateCulturalData()
    {
        float totalRespect = 0f;
        int totalInteractions = 0;
        int totalPrayers = 0;
        int totalActivities = 0;

        for (int i = 0; i < culturalInteractions.Length; i++)
        {
            totalInteractions += culturalInteractions[i];
            totalRespect += respectScores[i];
```

```
            totalPrayers += prayerParticipation[i];
            totalActivities += traditionalActivities[i];
        }

        // Calculate aggregated metrics (privacy-compliant averages)
        aggregatedMetrics[0] = totalInteractions > 0 ? totalRespect /
totalInteractions : 0f;
        aggregatedMetrics[1] = totalPrayers;
        aggregatedMetrics[2] = totalActivities;
        aggregatedMetrics[3] = totalInteractions;
    }
}

public static AnalyticsSystem Instance { get; private set; }

[Header("Privacy Settings")]
public bool enableAnalytics = true;
public bool anonymizeData = true;
public bool respectDoNotTrack = true;
public float anonymizationThreshold = 0.8f;

[Header("Cultural Analytics")]
public bool trackCulturalInteractions = true;
public bool trackPrayerParticipation = true;
public bool trackTraditionalActivities = true;
public bool trackLanguageUsage = true;

[Header("Performance Metrics")]
public bool trackPerformanceMetrics = true;
public bool trackDeviceInfo = true;
public bool trackGameplayMetrics = true;
```

```
private string sessionID;
private string anonymizedUserID;
private bool userConsented;
private DateTime sessionStartTime;

// Cultural tracking data
private int culturalInteractionsCount;
private float totalRespectScore;
private int prayerParticipationCount;
private int traditionalActivitiesCount;
private int dhivehiLanguageUsage;

void Awake()
{
    Instance = this;
    InitializeAnalyticsSystem();
}

void InitializeAnalyticsSystem()
{
    // Generate privacy-compliant identifiers
    sessionID = GenerateSessionID();
    anonymizedUserID = GenerateAnonymizedUserID();
    sessionStartTime = DateTime.Now;

    // Check user consent
    userConsented = CheckUserConsent();

    if (!userConsented)
    {
```

```
            Debug.Log("Analytics disabled - user consent required");
            return;
        }

        int dataPoints = 100;
        var gameplayMetrics = new NativeArray<float>(dataPoints,
Allocator.TempJob);
        var anonymizedData = new NativeArray<float>(dataPoints,
Allocator.TempJob);
        var privacyFlags = new NativeArray<bool>(dataPoints, Allocator.TempJob);

        // Initialize with sample data
        for (int i = 0; i < dataPoints; i++)
        {
            gameplayMetrics[i] = UnityEngine.Random.Range(0f, 1f);
        }

        var privacyJob = new PrivacyCompliantDataCollection
        {
            gameplayMetrics = gameplayMetrics,
            anonymizedData = anonymizedData,
            privacyFlags = privacyFlags,
            anonymizationThreshold = anonymizationThreshold,
            collectPersonalData = false
        };

        // Cultural metrics
        var culturalInteractions = new NativeArray<int>(50, Allocator.TempJob);
        var respectScores = new NativeArray<float>(50, Allocator.TempJob);
        var prayerParticipation = new NativeArray<int>(50, Allocator.TempJob);
        var traditionalActivities = new NativeArray<int>(50, Allocator.TempJob);
```

```
var aggregatedMetrics = new NativeArray<float>(4, Allocator.TempJob);

// Initialize cultural data
for (int i = 0; i < 50; i++)
{
    culturalInteractions[i] = UnityEngine.Random.Range(0, 5);
    respectScores[i] = UnityEngine.Random.Range(0f, 1f);
    prayerParticipation[i] = UnityEngine.Random.Range(0, 2);
    traditionalActivities[i] = UnityEngine.Random.Range(0, 3);
}

var culturalJob = new CulturalMetricsAggregation
{
    culturalInteractions = culturalInteractions,
    respectScores = respectScores,
    prayerParticipation = prayerParticipation,
    traditionalActivities = traditionalActivities,
    aggregatedMetrics = aggregatedMetrics
};

JobHandle privacyHandle = privacyJob.Schedule(dataPoints, 16);
JobHandle culturalHandle = culturalJob.Schedule(privacyHandle);
culturalHandle.Complete();

// Process results
ProcessAggregatedMetrics(aggregatedMetrics);

gameplayMetrics.Dispose();
anonymizedData.Dispose();
privacyFlags.Dispose();
culturalInteractions.Dispose();
```

```
        respectScores.Dispose();
        prayerParticipation.Dispose();
        traditionalActivities.Dispose();
        aggregatedMetrics.Dispose();

        StartAnalyticsCollection();
    }

    string GenerateSessionID()
    {
        // Generate anonymous session identifier
        return
$"session_{DateTime.Now.Ticks}_{UnityEngine.Random.Range(1000, 9999)}";
    }

    string GenerateAnonymizedUserID()
    {
        // Generate privacy-compliant user identifier (not device-specific)
        string timestamp = DateTime.Now.ToString("yyyyMMdd");
        string random = UnityEngine.Random.Range(10000, 99999).ToString();
        return $"user_{timestamp}_{random}";
    }

    bool CheckUserConsent()
    {
        // Check if user has consented to analytics
        // In production, this would check PlayerPrefs or a consent management
system
        return PlayerPrefs.GetInt("AnalyticsConsent", 0) == 1;
    }
```

```
    void ProcessAggregatedMetrics(NativeArray<float> metrics)
    {
        float avgRespect = metrics[0];
        int totalPrayers = (int)metrics[1];
        int totalActivities = (int)metrics[2];
        int totalInteractions = (int)metrics[3];

        Debug.Log($"Cultural Analytics - Avg Respect: {avgRespect:F2}, Prayers:
{totalPrayers}, Activities: {totalActivities}");
    }

    void StartAnalyticsCollection()
    {
        if (!enableAnalytics || !userConsented) return;

        // Start periodic data collection
        InvokeRepeating("CollectAnalytics", 30f, 300f); // Every 5 minutes
        InvokeRepeating("SendAnalyticsBatch", 60f, 600f); // Every 10 minutes
    }

    public void TrackCulturalInteraction(string interactionType, float respectScore)
    {
        if (!enableAnalytics || !trackCulturalInteractions) return;

        culturalInteractionsCount++;
        totalRespectScore += respectScore;

        // Log cultural interaction with privacy protection
        LogEvent("cultural_interaction", new Dictionary<string, object>
        {
            {"interaction_type", interactionType},
```

```
            {"respect_score", anonymizeData ? AnonymizeValue(respectScore) :
respectScore},
            {"timestamp", DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss")}
        });
    }

    public void TrackPrayerParticipation(string prayerType)
    {
        if (!enableAnalytics || !trackPrayerParticipation) return;

        prayerParticipationCount++;

        LogEvent("prayer_participation", new Dictionary<string, object>
        {
            {"prayer_type", prayerType},
            {"anonymized", true},
            {"timestamp", DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss")}
        });
    }

    public void TrackTraditionalActivity(string activityName)
    {
        if (!enableAnalytics || !trackTraditionalActivities) return;

        traditionalActivitiesCount++;

        LogEvent("traditional_activity", new Dictionary<string, object>
        {
            {"activity_name", activityName},
            {"cultural_significance", "high"},
            {"timestamp", DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss")}
```

```
    });
}

public void TrackDhivehiLanguageUsage(string context)
{
    if (!enableAnalytics || !trackLanguageUsage) return;

    dhivehiLanguageUsage++;

    LogEvent("dhivehi_usage", new Dictionary<string, object>
    {
        {"context", context},
        {"respectful_usage", true},
        {"timestamp", DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss")}
    });
}

public void TrackPerformanceMetric(string metricName, float value)
{
    if (!enableAnalytics || !trackPerformanceMetrics) return;

    LogEvent("performance_metric", new Dictionary<string, object>
    {
        {"metric_name", metricName},
        {"value", anonymizeData ? AnonymizeValue(value) : value},
        {"device_type", GetDeviceType()},
        {"timestamp", DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss")}
    });
}

float AnonymizeValue(float value)
```

```
{
    // Add noise to anonymize while preserving trends
    float noise = UnityEngine.Random.Range(-0.05f, 0.05f);
    return value + noise;
}

void LogEvent(string eventName, Dictionary<string, object> parameters)
{
    if (!enableAnalytics) return;

    // Create privacy-compliant event log
    string eventData = JsonUtility.ToJson(new AnalyticsEvent
    {
        session_id = sessionID,
        user_id = anonymizedUserID,
        event_name = eventName,
        parameters = parameters,
        timestamp = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss")
    });

    // Store locally (in production, would batch send to server)
    StoreEventLocally(eventData);
}

void StoreEventLocally(string eventData)
{
    // Store in PlayerPrefs for batch sending
    string key = $"analytics_event_{DateTime.Now.Ticks}";
    PlayerPrefs.SetString(key, eventData);
    PlayerPrefs.Save();
}
```

```
string GetDeviceType()
{
    // Classify device without collecting specific identifiers
    int memoryMB = SystemInfo.systemMemorySize;
    if (memoryMB >= 4096) return "high_end";
    else if (memoryMB >= 2048) return "mid_range";
    else return "low_end";
}

void CollectAnalytics()
{
    if (!enableAnalytics) return;

    // Collect batch of events
    List<string> events = GetStoredEvents();

    // Process and anonymize
    foreach (string eventData in events)
    {
        ProcessEventBatch(eventData);
    }
}

void SendAnalyticsBatch()
{
    if (!enableAnalytics) return;

    // In production, would send to analytics server
    // For now, just log summary
```

```
        Debug.Log($"Analytics Batch - Session: {sessionID}, Events:
{GetEventCount()}");
    }

    List<string> GetStoredEvents()
    {
        List<string> events = new List<string>();
        // Retrieve stored events from PlayerPrefs
        // Implementation would iterate through stored keys
        return events;
    }

    int GetEventCount()
    {
        // Count stored events
        return 0; // Simplified for this implementation
    }

    void ProcessEventBatch(string eventData)
    {
        // Process and validate event data
        // Ensure privacy compliance
        if (anonymizeData)
        {
            // Apply additional anonymization
        }
    }

    public void RequestDataDeletion()
    {
        // GDPR compliance - delete all user data
```

```
        PlayerPrefs.DeleteKey("AnalyticsConsent");

        // Clear stored events
        ClearStoredEvents();

        Debug.Log("User data deletion requested - all analytics data cleared");
    }

    void ClearStoredEvents()
    {
        // Clear all stored analytics events
        // Implementation would clear PlayerPrefs keys
    }

    public AnalyticsSummary GetSessionSummary()
    {
        return new AnalyticsSummary
        {
            session_duration_minutes = (float)(DateTime.Now -
sessionStartTime).TotalMinutes,
            cultural_interactions = culturalInteractionsCount,
            average_respect_score = culturalInteractionsCount > 0 ?
totalRespectScore / culturalInteractionsCount : 0f,
            prayer_participation = prayerParticipationCount,
            traditional_activities = traditionalActivitiesCount,
            dhivehi_usage = dhivehiLanguageUsage,
            session_id = sessionID
        };
    }

    [System.Serializable]
```

```
    public class AnalyticsEvent
    {
        public string session_id;
        public string user_id;
        public string event_name;
        public Dictionary<string, object> parameters;
        public string timestamp;
    }

    [System.Serializable]
    public class AnalyticsSummary
    {
        public float session_duration_minutes;
        public int cultural_interactions;
        public float average_respect_score;
        public int prayer_participation;
        public int traditional_activities;
        public int dhivehi_usage;
        public string session_id;
    }
}
```

32. MonetizationSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using UnityEngine.Purchasing;
```

```
[BurstCompile]
public class MonetizationSystem : MonoBehaviour, IStoreListener
{
    [BurstCompile]
    struct EthicalPricingCalculation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> basePrices;
        [ReadOnly] public NativeArray<int> culturalSignificance;
        [WriteOnly] public NativeArray<float> ethicalPrices;
        [WriteOnly] public NativeArray<bool> purchaseRecommendations;

        public float maxPriceMultiplier;
        public float culturalDiscountFactor;

        public void Execute(int index)
        {
            float basePrice = basePrices[index];
            int culturalValue = culturalSignificance[index];

            // Apply ethical pricing based on cultural value
            float ethicalPrice = CalculateEthicalPrice(basePrice, culturalValue);
            bool recommended = IsPurchaseRecommended(ethicalPrice,
culturalValue);

            ethicalPrices[index] = ethicalPrice;
            purchaseRecommendations[index] = recommended;
        }

        [BurstCompile]
        float CalculateEthicalPrice(float basePrice, int culturalValue)
```

```
    {
        // Higher cultural value = lower price (promote cultural appreciation)
        float culturalMultiplier = 1.0f - (culturalValue * culturalDiscountFactor);
        float ethicalPrice = basePrice * culturalMultiplier;

        // Ensure minimum price for sustainability
        float minimumPrice = basePrice * 0.3f;
        return math.max(ethicalPrice, minimumPrice);
    }

    [BurstCompile]
    bool IsPurchaseRecommended(float price, int culturalValue)
    {
        // Recommend purchases that promote cultural understanding
        return culturalValue >= 3 && price < 5.0f; // Under $5 for cultural items
    }
}

[BurstCompile]
struct NonPredatoryRecommendation : IJob
{
    public NativeArray<float> playerSpendingHistory;
    public NativeArray<float> timeSinceLastPurchase;
    public NativeArray<bool> purchaseRecommendations;
    public float totalSpent;

    public void Execute()
    {
        GenerateEthicalRecommendations();
    }
```

```
    [BurstCompile]
    void GenerateEthicalRecommendations()
    {
        for (int i = 0; i < playerSpendingHistory.Length; i++)
        {
            float spending = playerSpendingHistory[i];
            float timeSincePurchase = timeSinceLastPurchase[i];

            // Non-predatory recommendation logic
            bool shouldRecommend = ShouldRecommendPurchase(spending,
timeSincePurchase, totalSpent);
            purchaseRecommendations[i] = shouldRecommend;
        }
    }

    [BurstCompile]
    bool ShouldRecommendPurchase(float recentSpending, float
timeSincePurchase, float totalSpent)
    {
        // Ethical recommendation criteria
        bool canAfford = recentSpending < 10.0f; // Under $10 recent spending
        bool enoughTimePassed = timeSincePurchase > 86400f; // 24 hours
minimum
        bool notExcessive = totalSpent < 50.0f; // Under $50 total

        return canAfford && enoughTimePassed && notExcessive;
    }
}

public static MonetizationSystem Instance { get; private set; }
```

```
[Header("Ethical Pricing")]
public bool enableEthicalPricing = true;
public float culturalDiscountFactor = 0.2f;
public float maxItemPrice = 9.99f;
public float minCulturalItemPrice = 0.99f;

[Header("Cultural Items")]
public CulturalProduct[] culturalProducts;
public bool discountCulturalEducation = true;
public bool premiumCulturalContent = false; // No paywall for culture

[Header("Non-Predatory Features")]
public bool enableSpendingLimits = true;
public float dailySpendingLimit = 20.0f;
public float weeklySpendingLimit = 50.0f;
public bool enableCoolDownPeriods = true;
public float purchaseCoolDownHours = 24f;

[Header("Transparency")]
public bool showPriceBreakdown = true;
public bool showCulturalValue = true;
public bool enableReceiptEmails = true;

private IStoreController storeController;
private IExtensionProvider storeExtensionProvider;
private float totalPlayerSpending;
private DateTime lastPurchaseTime;
private float dailySpending;
private float weeklySpending;

[System.Serializable]
```

```
public class CulturalProduct
{
    public string productID;
    public string productName;
    public string dhivehiName;
    public string description;
    public ProductType productType;
    public float basePrice;
    public int culturalSignificance; // 1-5 scale
    public bool isEducational;
    public bool supportsLocalArtisans;
    public string culturalContext;
    public Sprite productIcon;

    public enum ProductType
    {
        Cosmetic,
        Educational,
        CulturalContent,
        Convenience,
        SupportLocal
    }
}

void Awake()
{
    Instance = this;
    InitializeMonetizationSystem();
}

void InitializeMonetizationSystem()
```

```
    {
        // Initialize Unity Purchasing
        if (storeController == null)
        {
            InitializePurchasing();
        }

        // Load player spending history
        LoadSpendingHistory();

        int productCount = 20;
        var basePrices = new NativeArray<float>(productCount,
Allocator.TempJob);
        var culturalSignificance = new NativeArray<int>(productCount,
Allocator.TempJob);
        var ethicalPrices = new NativeArray<float>(productCount,
Allocator.TempJob);
        var purchaseRecommendations = new NativeArray<bool>(productCount,
Allocator.TempJob);

        // Initialize product data
        for (int i = 0; i < productCount; i++)
        {
            basePrices[i] = UnityEngine.Random.Range(0.99f, 9.99f);
            culturalSignificance[i] = UnityEngine.Random.Range(1, 6);
        }

        var pricingJob = new EthicalPricingCalculation
        {
            basePrices = basePrices,
            culturalSignificance = culturalSignificance,
```

```
            ethicalPrices = ethicalPrices,
            purchaseRecommendations = purchaseRecommendations,
            maxPriceMultiplier = 2.0f,
            culturalDiscountFactor = culturalDiscountFactor
        };

        // Non-predatory recommendations
        var spendingHistory = new NativeArray<float>(10, Allocator.TempJob);
        var timeSincePurchases = new NativeArray<float>(10, Allocator.TempJob);
        var ethicalRecommendations = new NativeArray<bool>(10,
Allocator.TempJob);

        for (int i = 0; i < 10; i++)
        {
            spendingHistory[i] = UnityEngine.Random.Range(0f, 15f);
            timeSincePurchases[i] = UnityEngine.Random.Range(0f, 172800f); // 0-
48 hours
        }

        var recommendationJob = new NonPredatoryRecommendation
        {
            playerSpendingHistory = spendingHistory,
            timeSinceLastPurchase = timeSincePurchases,
            purchaseRecommendations = ethicalRecommendations,
            totalSpent = totalPlayerSpending
        };

        JobHandle pricingHandle = pricingJob.Schedule(productCount, 8);
        JobHandle recommendationHandle =
recommendationJob.Schedule(pricingHandle);
        recommendationHandle.Complete();
```

```
        // Process ethical pricing results
        ProcessEthicalPricing(ethicalPrices, purchaseRecommendations);

        basePrices.Dispose();
        culturalSignificance.Dispose();
        ethicalPrices.Dispose();
        purchaseRecommendations.Dispose();
        spendingHistory.Dispose();
        timeSincePurchases.Dispose();
        ethicalRecommendations.Dispose();

        ValidateEthicalConstraints();
    }

    void InitializePurchasing()
    {
        // Initialize Unity Purchasing
        var builder =
ConfigurationBuilder.Instance(StandardPurchasingModule.Instance());

        // Add cultural products
        foreach (var product in culturalProducts)
        {
            builder.AddProduct(product.productID, ProductType.Consumable, new
IDs
            {
                {product.productID, GooglePlay.Name},
                {product.productID, AppleAppStore.Name}
            });
        }
```

```
        UnityPurchasing.Initialize(this, builder);
    }

    public void OnInitialized(IStoreController controller, IExtensionProvider
extensions)
    {
        storeController = controller;
        storeExtensionProvider = extensions;
        Debug.Log("Ethical monetization system initialized");
    }

    public void OnInitializeFailed(InitializationFailureReason error)
    {
        Debug.LogError($"Monetization initialization failed: {error}");
    }

    void ProcessEthicalPricing(NativeArray<float> prices, NativeArray<bool>
recommendations)
    {
        // Apply ethical pricing to cultural products
        for (int i = 0; i < Mathf.Min(prices.Length, culturalProducts.Length); i++)
        {
            var product = culturalProducts[i];
            float ethicalPrice = prices[i];
            bool recommended = recommendations[i];

            // Ensure price respects ethical constraints
            ethicalPrice = Mathf.Clamp(ethicalPrice, minCulturalItemPrice,
maxItemPrice);
```

```
            Debug.Log($"{product.productName}: Ethical Price ${ethicalPrice:F2},
Recommended: {recommended}");
        }
    }

    void ValidateEthicalConstraints()
    {
        // Validate all products meet ethical standards
        foreach (var product in culturalProducts)
        {
            if (product.culturalSignificance >= 4 && product.basePrice > 5.0f)
            {
                Debug.LogWarning($"High cultural significance item
'{product.productName}' may be overpriced");
            }

            if (product.isEducational && product.basePrice > 3.0f)
            {
                Debug.LogWarning($"Educational cultural item
'{product.productName}' should be more accessible");
            }
        }
    }

    void LoadSpendingHistory()
    {
        // Load player spending history
        totalPlayerSpending = PlayerPrefs.GetFloat("TotalSpending", 0f);
        dailySpending = PlayerPrefs.GetFloat("DailySpending", 0f);
        weeklySpending = PlayerPrefs.GetFloat("WeeklySpending", 0f);
```

```
    string lastPurchaseStr = PlayerPrefs.GetString("LastPurchaseTime", "");
    if (!string.IsNullOrEmpty(lastPurchaseStr))
    {
        DateTime.TryParse(lastPurchaseStr, out lastPurchaseTime);
    }

    // Reset daily/weekly spending if needed
    CheckSpendingPeriodReset();
}

void CheckSpendingPeriodReset()
{
    DateTime now = DateTime.Now;

    // Reset daily spending
    if (now.Date > lastPurchaseTime.Date)
    {
        dailySpending = 0f;
    }

    // Reset weekly spending
    if (now.Date > lastPurchaseTime.Date.AddDays(7))
    {
        weeklySpending = 0f;
    }
}

public bool CanAffordPurchase(string productID, float price)
{
    if (!enableSpendingLimits) return true;
```

```
        // Check daily limit
        if (dailySpending + price > dailySpendingLimit)
        {
            Debug.Log("Daily spending limit would be exceeded");
            return false;
        }

        // Check weekly limit
        if (weeklySpending + price > weeklySpendingLimit)
        {
            Debug.Log("Weekly spending limit would be exceeded");
            return false;
        }

        // Check cool-down period
        if (enableCoolDownPeriods)
        {
            float hoursSinceLastPurchase = (float)(DateTime.Now -
lastPurchaseTime).TotalHours;
            if (hoursSinceLastPurchase < purchaseCoolDownHours)
            {
                Debug.Log($"Purchase cool-down active: {purchaseCoolDownHours -
hoursSinceLastPurchase:F1} hours remaining");
                return false;
            }
        }

        return true;
    }
```

```
    public void ProcessPurchase(string productID, float price, Action<bool>
callback)
    {
        if (!CanAffordPurchase(productID, price))
        {
            callback?.Invoke(false);
            return;
        }

        // Record purchase ethically
        RecordPurchase(productID, price);

        // Update spending tracking
        UpdateSpendingTracking(price);

        callback?.Invoke(true);
    }

    void RecordPurchase(string productID, float price)
    {
        var product = System.Array.Find(culturalProducts, p => p.productID ==
productID);

        if (product != null)
        {
            // Log ethical purchase
            Debug.Log($"Ethical purchase recorded: {product.productName} for
${price:F2}");

            // Track cultural impact
            if (product.supportsLocalArtisans)
```

```
            {
                Debug.Log($"Purchase supports local Maldivian artisans");
            }

            if (product.isEducational)
            {
                Debug.Log($"Educational purchase promotes cultural understanding");
            }
        }
    }

    void UpdateSpendingTracking(float amount)
    {
        totalPlayerSpending += amount;
        dailySpending += amount;
        weeklySpending += amount;
        lastPurchaseTime = DateTime.Now;

        // Save updated spending data
        PlayerPrefs.SetFloat("TotalSpending", totalPlayerSpending);
        PlayerPrefs.SetFloat("DailySpending", dailySpending);
        PlayerPrefs.SetFloat("WeeklySpending", weeklySpending);
        PlayerPrefs.SetString("LastPurchaseTime", lastPurchaseTime.ToString());
        PlayerPrefs.Save();
    }

    public PurchaseRecommendation GetPurchaseRecommendation(string
productID)
    {
        var product = System.Array.Find(culturalProducts, p => p.productID ==
productID);
```

```
        if (product == null) return new PurchaseRecommendation { recommended =
false, reason = "Product not found" };

        // Ethical recommendation logic
        bool recommended = ShouldRecommendPurchase(product);
        string reason = GetRecommendationReason(product, recommended);

        return new PurchaseRecommendation
        {
            recommended = recommended,
            reason = reason,
            ethicalPrice = CalculateEthicalPrice(product),
            culturalValue = product.culturalSignificance
        };
    }

    bool ShouldRecommendPurchase(CulturalProduct product)
    {
        // Non-predatory recommendation logic
        if (product.culturalSignificance >= 4) return true; // High cultural value
        if (product.isEducational) return true;
        if (product.supportsLocalArtisans) return true;

        // Check spending limits
        return CanAffordPurchase(product.productID, product.basePrice);
    }

    string GetRecommendationReason(CulturalProduct product, bool
recommended)
    {
```

```
        if (!recommended) return "Consider your budget and take time to decide";

        if (product.culturalSignificance >= 4) return "High cultural educational value";
        if (product.isEducational) return "Promotes cultural understanding";
        if (product.supportsLocalArtisans) return "Supports local Maldivian
community";

        return "Cultural enrichment opportunity";
    }

    float CalculateEthicalPrice(CulturalProduct product)
    {
        if (!enableEthicalPricing) return product.basePrice;

        float culturalMultiplier = 1.0f - (product.culturalSignificance *
culturalDiscountFactor);
        float ethicalPrice = product.basePrice * culturalMultiplier;

        // Ensure minimum price for sustainability
        float minimumPrice = product.isEducational ? minCulturalItemPrice :
product.basePrice * 0.5f;

        return Mathf.Max(ethicalPrice, minimumPrice);
    }

    public void OnPurchaseFailed(Product product, PurchaseFailureReason
failureReason)
    {
        Debug.LogError($"Purchase failed: {product.definition.id} -
{failureReason}");
```

```
        // Log ethical failure reason
        string ethicalReason = GetEthicalFailureReason(failureReason);
        Debug.Log($"Ethical purchase failure: {ethicalReason}");
    }

    string GetEthicalFailureReason(PurchaseFailureReason reason)
    {
        return reason switch
        {
            PurchaseFailureReason.PurchasingUnavailable => "Purchase system
temporarily unavailable",
            PurchaseFailureReason.ExistingPurchasePending => "Previous
purchase still processing",
            PurchaseFailureReason.ProductUnavailable => "Cultural content
currently unavailable",
            PurchaseFailureReason.UserCancelled => "Purchase cancelled by user",
            _ => "Purchase could not be completed"
        };
    }

    public SpendingSummary GetSpendingSummary()
    {
        return new SpendingSummary
        {
            total_spent = totalPlayerSpending,
            daily_spent = dailySpending,
            weekly_spent = weeklySpending,
            daily_limit = dailySpendingLimit,
            weekly_limit = weeklySpendingLimit,
            hours_since_last_purchase = (float)(DateTime.Now -
lastPurchaseTime).TotalHours,
```

```
        cool_down_hours = purchaseCoolDownHours
    };
}

[System.Serializable]
public class PurchaseRecommendation
{
    public bool recommended;
    public string reason;
    public float ethicalPrice;
    public int culturalValue;
}

[System.Serializable]
public class SpendingSummary
{
    public float total_spent;
    public float daily_spent;
    public float weekly_spent;
    public float daily_limit;
    public float weekly_limit;
    public float hours_since_last_purchase;
    public float cool_down_hours;
}

void OnDestroy()
{
    // Cleanup
    if (storeController != null)
    {
        // Store cleanup if needed
```

```
        }
    }
}
```

33. ReputationSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class ReputationSystem : MonoBehaviour
{
    [BurstCompile]
    struct ReputationCalculation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> actionScores;
        [ReadOnly] public NativeArray<int> actionTypes;
        [ReadOnly] public NativeArray<float> culturalContext;
        [WriteOnly] public NativeArray<float> reputationChanges;
        [WriteOnly] public NativeArray<float> finalReputation;

        public float currentReputation;
        public float culturalMultiplier;

        public void Execute(int index)
        {
            float actionScore = actionScores[index];
            int actionType = actionTypes[index];
```

```
            float context = culturalContext[index];

            // Calculate reputation change based on cultural appropriateness
            float reputationChange = CalculateReputationChange(actionScore,
actionType, context);
            float newReputation = currentReputation + reputationChange;

            reputationChanges[index] = reputationChange;
            finalReputation[index] = math.saturate(newReputation);
        }

        [BurstCompile]
        float CalculateReputationChange(float actionScore, int actionType, float
context)
        {
            // Cultural action types:
            // 1: Prayer participation
            // 2: Traditional activities
            // 3: Respectful behavior
            // 4: Cultural learning
            // 5: Community support

            float baseChange = actionScore * 0.1f;
            float typeMultiplier = actionType switch
            {
                1 => 1.5f, // Prayer (high cultural value)
                2 => 1.3f, // Traditional activities
                3 => 1.2f, // Respectful behavior
                4 => 1.1f, // Cultural learning
                5 => 1.4f, // Community support
                _ => 1.0f
```

```
        };

        float contextMultiplier = 1.0f + (context * 0.5f);

        return baseChange * typeMultiplier * contextMultiplier;
    }
}

[BurstCompile]
struct MaldivianCulturalReputation : IJob
{
    public NativeArray<float> islandReputation;
    public NativeArray<float> communityStanding;
    public NativeArray<float> religiousRespect;
    public NativeArray<float> traditionalKnowledge;
    public NativeArray<float> familyReputation;

    public NativeArray<float> overallCulturalReputation;

    public void Execute()
    {
        CalculateMaldivianCulturalStanding();
    }

    [BurstCompile]
    void CalculateMaldivianCulturalStanding()
    {
        // Weighted average for Maldivian cultural reputation
        float islandWeight = 0.25f;
        float communityWeight = 0.25f;
        float religiousWeight = 0.20f;
```

```
        float knowledgeWeight = 0.15f;
        float familyWeight = 0.15f;

        float totalReputation = 0f;

        for (int i = 0; i < overallCulturalReputation.Length; i++)
        {
            float islandRep = islandReputation[i];
            float communityRep = communityStanding[i];
            float religiousRep = religiousRespect[i];
            float knowledgeRep = traditionalKnowledge[i];
            float familyRep = familyReputation[i];

            totalReputation = (islandRep * islandWeight) +
                              (communityRep * communityWeight) +
                              (religiousRep * religiousWeight) +
                              (knowledgeRep * knowledgeWeight) +
                              (familyRep * familyWeight);

            overallCulturalReputation[i] = math.saturate(totalReputation);
        }
    }
}

public static ReputationSystem Instance { get; private set; }

[Header("Reputation Settings")]
public float startingReputation = 0.5f;
public float maxReputation = 1.0f;
public float minReputation = 0.0f;
public float reputationDecayRate = 0.001f;
```

```
[Header("Cultural Reputation")]
public bool enableCulturalReputation = true;
public float culturalReputationWeight = 0.4f;
public bool respectIslandCommunities = true;
public bool trackReligiousReputation = true;

[Header("Reputation Categories")]
public ReputationCategory[] reputationCategories;

private float overallReputation;
private float[,] islandReputation; // [islandID, category]
private float[] communityReputation;
private float religiousReputation;
private float traditionalKnowledge;
private float familyReputationScore;

[System.Serializable]
public class ReputationCategory
{
    public string categoryName;
    public string dhivehiName;
    public float currentScore;
    public float maxScore = 1.0f;
    public float weight = 1.0f;
    public bool isCultural;
    public Color reputationColor;

    public enum CategoryType
    {
        ReligiousRespect,
```

```
        CommunityStanding,
        TraditionalKnowledge,
        FamilyReputation,
        IslandLoyalty,
        BusinessIntegrity,
        CulturalPreservation,
        EnvironmentalResponsibility,
        SocialHarmony,
        EducationalContribution
    }
}

void Awake()
{
    Instance = this;
    InitializeReputationSystem();
}

void InitializeReputationSystem()
{
    // Initialize overall reputation
    overallReputation = startingReputation;

    // Initialize reputation tracking arrays
    islandReputation = new float[41, 10]; // 41 islands, 10 reputation categories
    communityReputation = new float[83]; // 83 communities/gangs

    int reputationActions = 100;
    var actionScores = new NativeArray<float>(reputationActions,
Allocator.TempJob);
```

```
        var actionTypes = new NativeArray<int>(reputationActions,
Allocator.TempJob);
        var culturalContext = new NativeArray<float>(reputationActions,
Allocator.TempJob);
        var reputationChanges = new NativeArray<float>(reputationActions,
Allocator.TempJob);
        var finalReputation = new NativeArray<float>(reputationActions,
Allocator.TempJob);

        // Initialize action data
        for (int i = 0; i < reputationActions; i++)
        {
            actionScores[i] = UnityEngine.Random.Range(-1f, 1f);
            actionTypes[i] = UnityEngine.Random.Range(1, 6);
            culturalContext[i] = UnityEngine.Random.Range(0f, 1f);
        }

        var reputationJob = new ReputationCalculation
        {
            actionScores = actionScores,
            actionTypes = actionTypes,
            culturalContext = culturalContext,
            reputationChanges = reputationChanges,
            finalReputation = finalReputation,
            currentReputation = overallReputation,
            culturalMultiplier = culturalReputationWeight
        };

        // Maldivian cultural reputation
        var islandRep = new NativeArray<float>(41, Allocator.TempJob);
        var communityRep = new NativeArray<float>(83, Allocator.TempJob);
```

```
        var religiousRep = new NativeArray<float>(1, Allocator.TempJob);
        var traditionalRep = new NativeArray<float>(1, Allocator.TempJob);
        var familyRep = new NativeArray<float>(1, Allocator.TempJob);
        var overallCulturalRep = new NativeArray<float>(1, Allocator.TempJob);

        // Initialize cultural reputation data
        for (int i = 0; i < 41; i++)
        {
            islandRep[i] = UnityEngine.Random.Range(0.3f, 0.7f);
        }
        for (int i = 0; i < 83; i++)
        {
            communityRep[i] = UnityEngine.Random.Range(0.2f, 0.8f);
        }
        religiousRep[0] = UnityEngine.Random.Range(0.4f, 0.9f);
        traditionalRep[0] = UnityEngine.Random.Range(0.3f, 0.8f);
        familyRep[0] = UnityEngine.Random.Range(0.5f, 0.9f);

        var culturalJob = new MaldivianCulturalReputation
        {
            islandReputation = islandRep,
            communityStanding = communityRep,
            religiousRespect = religiousRep,
            traditionalKnowledge = traditionalRep,
            familyReputation = familyRep,
            overallCulturalReputation = overallCulturalRep
        };

        JobHandle reputationHandle = reputationJob.Schedule(reputationActions,
16);
        JobHandle culturalHandle = culturalJob.Schedule(reputationHandle);
```

```
        culturalHandle.Complete();

        // Update overall cultural reputation
        religiousReputation = religiousRep[0];
        traditionalKnowledge = traditionalRep[0];
        familyReputationScore = familyRep[0];

        religiousRep.Dispose();
        traditionalRep.Dispose();
        familyRep.Dispose();
        overallCulturalRep.Dispose();
        actionScores.Dispose();
        actionTypes.Dispose();
        culturalContext.Dispose();
        reputationChanges.Dispose();
        finalReputation.Dispose();
        islandRep.Dispose();
        communityRep.Dispose();

        StartReputationMonitoring();
    }

    void StartReputationMonitoring()
    {
        // Monitor reputation changes
        InvokeRepeating("UpdateReputationDecay", 60f, 300f); // Every 5 minutes
        InvokeRepeating("UpdateCulturalReputation", 30f, 180f); // Every 3 minutes
    }

    public void ModifyReputation(float change, string category, int islandID = -1, int
communityID = -1)
```

```
    {
        // Apply reputation change with cultural sensitivity
        float modifiedChange = ApplyCulturalReputationModifiers(change,
category);

        // Update overall reputation
        overallReputation = Mathf.Clamp(overallReputation + modifiedChange,
minReputation, maxReputation);

        // Update specific reputation categories
        if (islandID >= 0 && islandID < 41)
        {
            UpdateIslandReputation(islandID, category, modifiedChange);
        }

        if (communityID >= 0 && communityID < 83)
        {
            UpdateCommunityReputation(communityID, modifiedChange);
        }

        // Update cultural reputation if applicable
        if (IsCulturalReputationCategory(category))
        {
            UpdateCulturalReputation(category, modifiedChange);
        }

        // Trigger reputation change events
        OnReputationChanged(category, modifiedChange);
    }

    float ApplyCulturalReputationModifiers(float baseChange, string category)
```

```
{
    if (!enableCulturalReputation) return baseChange;

    // Apply cultural context to reputation changes
    float culturalMultiplier = category.ToLower() switch
    {
        "prayer_participation" => 1.5f,
        "traditional_respect" => 1.3f,
        "community_support" => 1.2f,
        "cultural_learning" => 1.1f,
        "religious_disrespect" => -1.5f,
        "cultural_insensitivity" => -1.3f,
        _ => 1.0f
    };

    return baseChange * culturalMultiplier;
}

void UpdateIslandReputation(int islandID, string category, float change)
{
    // Find reputation category index
    int categoryIndex = GetReputationCategoryIndex(category);
    if (categoryIndex >= 0)
    {
        islandReputation[islandID, categoryIndex] = Mathf.Clamp(
            islandReputation[islandID, categoryIndex] + change,
            minReputation,
            maxReputation
        );
    }
}
```

```
    void UpdateCommunityReputation(int communityID, float change)
    {
        communityReputation[communityID] = Mathf.Clamp(
            communityReputation[communityID] + change,
            minReputation,
            maxReputation
        );
    }

    void UpdateCulturalReputation(string category, float change)
    {
        switch (category.ToLower())
        {
            case "religious_respect":
                religiousReputation = Mathf.Clamp(religiousReputation + change,
minReputation, maxReputation);
                break;
            case "traditional_knowledge":
                traditionalKnowledge = Mathf.Clamp(traditionalKnowledge + change,
minReputation, maxReputation);
                break;
            case "family_reputation":
                familyReputationScore = Mathf.Clamp(familyReputationScore +
change, minReputation, maxReputation);
                break;
        }
    }

    bool IsCulturalReputationCategory(string category)
    {
```

```
        string[] culturalCategories = {
            "religious_respect", "traditional_knowledge", "cultural_preservation",
            "community_standing", "island_loyalty", "family_reputation"
        };

        return System.Array.Exists(culturalCategories, c => c ==
category.ToLower());
    }

    int GetReputationCategoryIndex(string category)
    {
        for (int i = 0; i < reputationCategories.Length; i++)
        {
            if (reputationCategories[i].categoryName.ToLower() ==
category.ToLower())
                return i;
        }
        return -1;
    }

    void OnReputationChanged(string category, float change)
    {
        Debug.Log($"Reputation changed: {category} by {change:F3}. New overall:
{overallReputation:F3}");

        // Check for reputation milestones
        CheckReputationMilestones();

        // Update UI if needed
        UpdateReputationUI();
    }
```

```
    void CheckReputationMilestones()
    {
        if (overallReputation >= 0.8f)
        {
            Debug.Log("Reputation milestone achieved: Highly Respected");
            // Trigger positive events
        }
        else if (overallReputation <= 0.2f)
        {
            Debug.Log("Reputation warning: Poor standing in community");
            // Trigger reputation recovery opportunities
        }
    }

    void UpdateReputationUI()
    {
        // Update UI elements to reflect current reputation
        // Implementation would update specific UI components
    }

    void UpdateReputationDecay()
    {
        // Natural reputation decay over time
        if (overallReputation > 0.5f)
        {
            overallReputation = Mathf.Max(overallReputation - reputationDecayRate,
0.5f);
        }
        else if (overallReputation < 0.5f)
        {
```

```
            overallReputation = Mathf.Min(overallReputation + reputationDecayRate,
0.5f);
        }

        // Decay island reputations
        for (int i = 0; i < 41; i++)
        {
            for (int j = 0; j < 10; j++)
            {
                if (islandReputation[i, j] > 0.5f)
                    islandReputation[i, j] = Mathf.Max(islandReputation[i, j] -
reputationDecayRate * 0.5f, 0.5f);
            }
        }
    }

    void UpdateCulturalReputation()
    {
        // Update overall cultural reputation based on components
        float culturalRep = (religiousReputation + traditionalKnowledge +
familyReputationScore) / 3.0f;

        // Influence overall reputation
        if (enableCulturalReputation)
        {
            overallReputation = Mathf.Lerp(overallReputation, culturalRep,
culturalReputationWeight * 0.1f);
        }
    }

    public float GetOverallReputation()
```

```
    {
        return overallReputation;
    }

    public float GetIslandReputation(int islandID, string category)
    {
        int categoryIndex = GetReputationCategoryIndex(category);
        if (islandID >= 0 && islandID < 41 && categoryIndex >= 0)
        {
            return islandReputation[islandID, categoryIndex];
        }
        return 0.5f;
    }

    public float GetCommunityReputation(int communityID)
    {
        if (communityID >= 0 && communityID < 83)
        {
            return communityReputation[communityID];
        }
        return 0.5f;
    }

    public float GetCulturalReputation()
    {
        return (religiousReputation + traditionalKnowledge + familyReputationScore)
/ 3.0f;
    }

    public string GetReputationLevel()
    {
```

```
    return overallReputation switch
    {
        >= 0.9f => "Legendary",
        >= 0.8f => "Highly Respected",
        >= 0.7f => "Well Regarded",
        >= 0.6f => "Respected",
        >= 0.5f => "Average",
        >= 0.4f => "Questionable",
        >= 0.3f => "Poor",
        >= 0.2f => "Disreputable",
        _ => "Notorious"
    };
}

public string GetDhivehiReputationTitle()
{
    return GetReputationLevel() switch
    {
        "Legendary" => "fiÜfi¶fiçfiØfiÇfi∞fiçfi™ fiàfi®fiçfi™",
        "Highly Respected" => "fiÅfi¶fiÅfi©fiÜfi™ fiàfi®fiçfi™",
        "Well Regarded" => "fiàfi¶fiÅfi©fiÜfi™ fiàfi®fiçfi™",
        "Respected" => "fiàfi®fiçfi™",
        "Average" => "fiàfi¶fiÅfi©fiÜfi™",
        "Questionable" => "fiàfi¶fiÅfi©fiÜfi™ fiàfi¶fiÅfi©fiÜfi™",
        "Poor" => "fiàfi¶fiÅfi©fiÜfi™ fiàfi¶fiÅfi©fiÜfi™",
        "Disreputable" => "fiàfi¶fiÅfi©fiÜfi™ fiàfi¶fiÅfi©fiÜfi™",
        _ => "fiàfi¶fiÅfi©fiÜfi™ fiàfi¶fiÅfi©fiÜfi™"
    };
}

public ReputationSnapshot GetReputationSnapshot()
```

```
{
    return new ReputationSnapshot
    {
        overall_reputation = overallReputation,
        cultural_reputation = GetCulturalReputation(),
        religious_reputation = religiousReputation,
        traditional_knowledge = traditionalKnowledge,
        family_reputation = familyReputationScore,
        reputation_level = GetReputationLevel(),
        dhivehi_title = GetDhivehiReputationTitle(),
        islands_visited = GetIslandsWithHighReputation(),
        communities_respected = GetRespectedCommunities()
    };
}

int GetIslandsWithHighReputation()
{
    int count = 0;
    for (int i = 0; i < 41; i++)
    {
        if (GetAverageIslandReputation(i) > 0.7f) count++;
    }
    return count;
}

int GetRespectedCommunities()
{
    int count = 0;
    for (int i = 0; i < 83; i++)
    {
        if (communityReputation[i] > 0.7f) count++;
```

```
        }
        return count;
    }

    float GetAverageIslandReputation(int islandID)
    {
        float total = 0f;
        for (int i = 0; i < 10; i++)
        {
            total += islandReputation[islandID, i];
        }
        return total / 10.0f;
    }

    [System.Serializable]
    public class ReputationSnapshot
    {
        public float overall_reputation;
        public float cultural_reputation;
        public float religious_reputation;
        public float traditional_knowledge;
        public float family_reputation;
        public string reputation_level;
        public string dhivehi_title;
        public int islands_visited;
        public int communities_respected;
    }
}
```

34. SkillSystem.cs

```
using UnityEngine;
```

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class SkillSystem : MonoBehaviour
{
    [BurstCompile]
    struct SkillProgressionCalculation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> skillExperience;
        [ReadOnly] public NativeArray<int> skillLevels;
        [ReadOnly] public NativeArray<float> learningRates;
        [WriteOnly] public NativeArray<int> newSkillLevels;
        [WriteOnly] public NativeArray<bool> levelUpTriggers;

        public float experienceMultiplier;
        public int maxSkillLevel;

        public void Execute(int index)
        {
            float experience = skillExperience[index];
            int currentLevel = skillLevels[index];
            float learningRate = learningRates[index];

            // Calculate skill progression
            int newLevel = CalculateNewLevel(experience, currentLevel,
learningRate);
            bool leveledUp = newLevel > currentLevel;
```

```
            newSkillLevels[index] = newLevel;
            levelUpTriggers[index] = leveledUp;
        }

        [BurstCompile]
        int CalculateNewLevel(float experience, int currentLevel, float learningRate)
        {
            // Experience curve: more experience needed at higher levels
            float experienceNeeded = GetExperienceForLevel(currentLevel + 1) /
learningRate;

            if (experience >= experienceNeeded && currentLevel < maxSkillLevel)
            {
                return currentLevel + 1;
            }

            return currentLevel;
        }

        [BurstCompile]
        float GetExperienceForLevel(int level)
        {
            // Exponential experience curve
            return 100f * math.pow(1.5f, level - 1);
        }
    }

    [BurstCompile]
    struct MaldivianCulturalSkills : IJob
    {
        public NativeArray<float> fishingSkill;
```

```
public NativeArray<float> navigationSkill;
public NativeArray<float> culturalKnowledge;
public NativeArray<float> religiousUnderstanding;
public NativeArray<float> traditionalCrafts;
public NativeArray<float> dhivehiLanguage;
public NativeArray<float> communityHarmony;
public NativeArray<float> environmentalRespect;

public NativeArray<float> overallCulturalCompetency;

public void Execute()
{
    CalculateCulturalCompetency();
}

[BurstCompile]
void CalculateCulturalCompetency()
{
    for (int i = 0; i < overallCulturalCompetency.Length; i++)
    {
        // Weighted average of cultural skills
        float fishing = fishingSkill[i] * 0.15f;
        float navigation = navigationSkill[i] * 0.10f;
        float knowledge = culturalKnowledge[i] * 0.20f;
        float religious = religiousUnderstanding[i] * 0.20f;
        float crafts = traditionalCrafts[i] * 0.10f;
        float language = dhivehiLanguage[i] * 0.15f;
        float harmony = communityHarmony[i] * 0.05f;
        float environment = environmentalRespect[i] * 0.05f;
```

```
            overallCulturalCompetency[i] = fishing + navigation + knowledge +
religious +
                                crafts + language + harmony + environment;
        }
    }
}

public static SkillSystem Instance { get; private set; }

[Header("Skill Settings")]
public int maxSkillLevel = 100;
public float experienceMultiplier = 1.0f;
public bool enableSkillDecay = false;
public float skillDecayRate = 0.001f;

[Header("Maldivian Cultural Skills")]
public bool enableTraditionalSkills = true;
public bool enableCulturalSkills = true;
public bool enableReligiousSkills = true;

[Header("Skill Categories")]
public PlayerSkill[] playerSkills;
public CulturalSkill[] culturalSkills;

private float[] skillExperience;
private int[] skillLevels;
private float[] skillLearningRates;
private bool[] skillUnlocked;

[System.Serializable]
public class PlayerSkill
```

```
{
    public string skillName;
    public string skillDescription;
    public SkillCategory category;
    public int currentLevel;
    public float currentExperience;
    public float experienceToNextLevel;
    public Sprite skillIcon;
    public bool isUnlocked;
    public string[] levelDescriptions;
    public float[] levelBonuses;

    public enum SkillCategory
    {
        Physical,
        Mental,
        Social,
        Cultural,
        Technical,
        Survival,
        Creative,
        Spiritual,
        Leadership,
        Knowledge
    }
}

[System.Serializable]
public class CulturalSkill
{
    public string skillName;
```

```
    public string dhivehiName;
    public CulturalSkillType skillType;
    public int currentLevel;
    public float culturalSignificance;
    public string traditionalContext;
    public bool requiresMentor;
    public string[] learningSteps;
    public float[] levelRequirements;

    public enum CulturalSkillType
    {
        TraditionalFishing,
        BoduberuDrumming,
        DhivehiLanguage,
        NavigationSkills,
        PrayerRecitation,
        CulturalStorytelling,
        TraditionalCrafts,
        CommunityHarmony,
        EnvironmentalRespect,
        ReligiousUnderstanding
    }
}

void Awake()
{
    Instance = this;
    InitializeSkillSystem();
}

void InitializeSkillSystem()
```

```
    {
        // Initialize skill tracking arrays
        int totalSkills = playerSkills.Length + culturalSkills.Length;
        skillExperience = new float[totalSkills];
        skillLevels = new int[totalSkills];
        skillLearningRates = new float[totalSkills];
        skillUnlocked = new bool[totalSkills];

        // Initialize skill data
        for (int i = 0; i < totalSkills; i++)
        {
            skillExperience[i] = 0f;
            skillLevels[i] = 1;
            skillLearningRates[i] = 1.0f;
            skillUnlocked[i] = i < playerSkills.Length ? playerSkills[i].isUnlocked :
false;
        }

        int skillCalculations = totalSkills;
        var skillExp = new NativeArray<float>(skillCalculations, Allocator.TempJob);
        var skillLvls = new NativeArray<int>(skillCalculations, Allocator.TempJob);
        var learningRates = new NativeArray<float>(skillCalculations,
Allocator.TempJob);
        var newSkillLvls = new NativeArray<int>(skillCalculations,
Allocator.TempJob);
        var levelUpTriggers = new NativeArray<bool>(skillCalculations,
Allocator.TempJob);

        // Initialize skill data for job
        for (int i = 0; i < skillCalculations; i++)
        {
```

```
            skillExp[i] = skillExperience[i];
            skillLvls[i] = skillLevels[i];
            learningRates[i] = skillLearningRates[i];
        }

        var progressionJob = new SkillProgressionCalculation
        {
            skillExperience = skillExp,
            skillLevels = skillLvls,
            learningRates = learningRates,
            newSkillLevels = newSkillLvls,
            levelUpTriggers = levelUpTriggers,
            experienceMultiplier = experienceMultiplier,
            maxSkillLevel = maxSkillLevel
        };

        // Maldivian cultural skills
        var fishingSkill = new NativeArray<float>(1, Allocator.TempJob);
        var navigationSkill = new NativeArray<float>(1, Allocator.TempJob);
        var culturalKnowledge = new NativeArray<float>(1, Allocator.TempJob);
        var religiousUnderstanding = new NativeArray<float>(1,
Allocator.TempJob);
        var traditionalCrafts = new NativeArray<float>(1, Allocator.TempJob);
        var dhivehiLanguage = new NativeArray<float>(1, Allocator.TempJob);
        var communityHarmony = new NativeArray<float>(1, Allocator.TempJob);
        var environmentalRespect = new NativeArray<float>(1, Allocator.TempJob);
        var overallCompetency = new NativeArray<float>(1, Allocator.TempJob);

        // Initialize cultural skill levels
        fishingSkill[0] =
GetCulturalSkillLevel(CulturalSkill.CulturalSkillType.TraditionalFishing);
```

```
        navigationSkill[0] =
GetCulturalSkillLevel(CulturalSkill.CulturalSkillType.NavigationSkills);
        culturalKnowledge[0] =
GetCulturalSkillLevel(CulturalSkill.CulturalSkillType.CulturalStorytelling);
        religiousUnderstanding[0] =
GetCulturalSkillLevel(CulturalSkill.CulturalSkillType.ReligiousUnderstanding);
        traditionalCrafts[0] =
GetCulturalSkillLevel(CulturalSkill.CulturalSkillType.TraditionalCrafts);
        dhivehiLanguage[0] =
GetCulturalSkillLevel(CulturalSkill.CulturalSkillType.DhivehiLanguage);
        communityHarmony[0] =
GetCulturalSkillLevel(CulturalSkill.CulturalSkillType.CommunityHarmony);
        environmentalRespect[0] =
GetCulturalSkillLevel(CulturalSkill.CulturalSkillType.EnvironmentalRespect);

        var culturalJob = new MaldivianCulturalSkills
        {
            fishingSkill = fishingSkill,
            navigationSkill = navigationSkill,
            culturalKnowledge = culturalKnowledge,
            religiousUnderstanding = religiousUnderstanding,
            traditionalCrafts = traditionalCrafts,
            dhivehiLanguage = dhivehiLanguage,
            communityHarmony = communityHarmony,
            environmentalRespect = environmentalRespect,
            overallCulturalCompetency = overallCompetency
        };

        JobHandle progressionHandle = progressionJob.Schedule(skillCalculations,
8);
        JobHandle culturalHandle = culturalJob.Schedule(progressionHandle);
```

```
    culturalHandle.Complete();

    // Update cultural competency
    float culturalScore = overallCompetency[0];
    Debug.Log($"Overall Cultural Competency: {culturalScore:F2}");

    skillExp.Dispose();
    skillLvls.Dispose();
    learningRates.Dispose();
    newSkillLvls.Dispose();
    levelUpTriggers.Dispose();
    fishingSkill.Dispose();
    navigationSkill.Dispose();
    culturalKnowledge.Dispose();
    religiousUnderstanding.Dispose();
    traditionalCrafts.Dispose();
    dhivehiLanguage.Dispose();
    communityHarmony.Dispose();
    environmentalRespect.Dispose();
    overallCompetency.Dispose();

    StartSkillDevelopment();
}

void StartSkillDevelopment()
{
    // Monitor skill development
    InvokeRepeating("UpdateSkillDecay", 300f, 600f); // Every 10 minutes
    InvokeRepeating("CheckSkillMilestones", 60f, 300f); // Every 5 minutes
}
```

```
    public void GainSkillExperience(string skillName, float experience, bool
isCultural = false)
    {
        int skillIndex = GetSkillIndex(skillName);
        if (skillIndex < 0) return;

        // Apply cultural bonus if applicable
        float modifiedExperience = experience * experienceMultiplier;
        if (isCultural && enableCulturalSkills)
        {
            modifiedExperience *= 1.2f; // Cultural learning bonus
        }

        // Add experience to skill
        skillExperience[skillIndex] += modifiedExperience;

        // Check for level up
        CheckSkillLevelUp(skillIndex);

        // Log skill development
        Debug.Log($"Skill '{skillName}' gained {modifiedExperience:F2}
experience");
    }

    void CheckSkillLevelUp(int skillIndex)
    {
        if (skillIndex >= skillLevels.Length) return;

        float currentExp = skillExperience[skillIndex];
        int currentLevel = skillLevels[skillIndex];
```

```
    // Calculate experience needed for next level
    float expNeeded = GetExperienceForLevel(currentLevel + 1);

    if (currentExp >= expNeeded && currentLevel < maxSkillLevel)
    {
        // Level up!
        skillLevels[skillIndex]++;
        OnSkillLevelUp(skillIndex, currentLevel + 1);
    }
}

void OnSkillLevelUp(int skillIndex, int newLevel)
{
    string skillName = GetSkillName(skillIndex);
    Debug.Log($"SKILL LEVEL UP: {skillName} reached level {newLevel}!");

    // Trigger level up effects
    if (skillIndex < playerSkills.Length)
    {
        var skill = playerSkills[skillIndex];
        skill.currentLevel = newLevel;

        // Apply level bonuses
        ApplySkillBonuses(skill, newLevel);
    }
    else
    {
        int culturalIndex = skillIndex - playerSkills.Length;
        if (culturalIndex < culturalSkills.Length)
        {
            var culturalSkill = culturalSkills[culturalIndex];
```

```
            culturalSkill.currentLevel = newLevel;

            // Apply cultural skill bonuses
            ApplyCulturalSkillBonuses(culturalSkill, newLevel);
        }
    }

    // Show level up notification
    ShowSkillLevelUpNotification(skillName, newLevel);
}

void ApplySkillBonuses(PlayerSkill skill, int level)
{
    if (level < skill.levelBonuses.Length)
    {
        float bonus = skill.levelBonuses[level - 1];
        Debug.Log($"{skill.skillName} bonus applied: {bonus:F2}");

        // Apply bonus to relevant game systems
        ApplyBonusToSystem(skill.category, bonus);
    }
}

void ApplyCulturalSkillBonuses(CulturalSkill skill, int level)
{
    Debug.Log($"Cultural skill '{skill.skillName}' advanced to level {level}");

    // Apply cultural skill benefits
    switch (skill.skillType)
    {
        case CulturalSkill.CulturalSkillType.DhivehiLanguage:
```

```
            // Improve dialogue options
            DialogueSystem.Instance?.UpdateLanguageProficiency(level);
            break;

        case CulturalSkill.CulturalSkillType.TraditionalFishing:
            // Improve fishing success rate
            FishingSystem.Instance?.UpdateFishingSkill(level);
            break;

        case CulturalSkill.CulturalSkillType.ReligiousUnderstanding:
            // Improve prayer time accuracy
            PrayerTimeSystem.Instance?.UpdateUnderstandingLevel(level);
            break;
    }
}

void ApplyBonusToSystem(PlayerSkill.SkillCategory category, float bonus)
{
    // Apply skill bonuses to relevant game systems
    switch (category)
    {
        case PlayerSkill.SkillCategory.Physical:
            // Apply to movement, stamina, etc.
            break;

        case PlayerSkill.SkillCategory.Social:
            // Apply to dialogue, reputation, etc.
            ReputationSystem.Instance?.ApplySkillBonus(bonus);
            break;

        case PlayerSkill.SkillCategory.Cultural:
```

```
            // Apply to cultural interactions
            if (enableCulturalSkills)
            {
                culturalReputationWeight += bonus * 0.1f;
            }
            break;
        }
    }

    void ShowSkillLevelUpNotification(string skillName, int level)
    {
        // Create culturally appropriate level up notification
        string message = $"fiÜfi¶fiçfiØfiÇfi∞ fiàfi®fiçfi™! {skillName} -
fiÜfi™fiÉfi™fiÇfi∞ {level}"; // "Skill improved! [Skill] - Level [X]"

        // Show notification through UI system
        UISystem.Instance?.ShowCulturalNotification(message, 3.0f);
    }

    public bool IsSkillUnlocked(string skillName)
    {
        int skillIndex = GetSkillIndex(skillName);
        return skillIndex >= 0 && skillUnlocked[skillIndex];
    }

    public void UnlockSkill(string skillName)
    {
        int skillIndex = GetSkillIndex(skillName);
        if (skillIndex >= 0)
        {
            skillUnlocked[skillIndex] = true;
```

```
        if (skillIndex < playerSkills.Length)
        {
            playerSkills[skillIndex].isUnlocked = true;
        }

        Debug.Log($"Skill unlocked: {skillName}");
    }
}

public float GetSkillLevel(string skillName)
{
    int skillIndex = GetSkillIndex(skillName);
    if (skillIndex >= 0)
    {
        return skillLevels[skillIndex];
    }
    return 0f;
}

public float GetSkillExperience(string skillName)
{
    int skillIndex = GetSkillIndex(skillName);
    if (skillIndex >= 0)
    {
        return skillExperience[skillIndex];
    }
    return 0f;
}

public float GetSkillProgress(string skillName)
```

```
{
    int skillIndex = GetSkillIndex(skillName);
    if (skillIndex >= 0)
    {
        int currentLevel = skillLevels[skillIndex];
        float currentExp = skillExperience[skillIndex];
        float expNeeded = GetExperienceForLevel(currentLevel + 1);
        float expCurrentLevel = GetExperienceForLevel(currentLevel);

        return (currentExp - expCurrentLevel) / (expNeeded - expCurrentLevel);
    }
    return 0f;
}

int GetSkillIndex(string skillName)
{
    // Search in player skills
    for (int i = 0; i < playerSkills.Length; i++)
    {
        if (playerSkills[i].skillName == skillName)
            return i;
    }

    // Search in cultural skills
    for (int i = 0; i < culturalSkills.Length; i++)
    {
        if (culturalSkills[i].skillName == skillName)
            return playerSkills.Length + i;
    }

    return -1;
```

```
}

string GetSkillName(int skillIndex)
{
    if (skillIndex < playerSkills.Length)
    {
        return playerSkills[skillIndex].skillName;
    }
    else
    {
        int culturalIndex = skillIndex - playerSkills.Length;
        if (culturalIndex < culturalSkills.Length)
        {
            return culturalSkills[culturalIndex].skillName;
        }
    }
    return "Unknown Skill";
}

float GetExperienceForLevel(int level)
{
    // Exponential experience curve
    return 100f * Mathf.Pow(1.5f, level - 1);
}

int GetCulturalSkillLevel(CulturalSkill.CulturalSkillType skillType)
{
    for (int i = 0; i < culturalSkills.Length; i++)
    {
        if (culturalSkills[i].skillType == skillType)
        {
```

```
            return culturalSkills[i].currentLevel;
        }
    }
    return 1;
}

void UpdateSkillDecay()
{
    if (!enableSkillDecay) return;

    // Gradual skill decay (very slow)
    for (int i = 0; i < skillExperience.Length; i++)
    {
        skillExperience[i] = Mathf.Max(0f, skillExperience[i] - skillDecayRate);
    }
}

void CheckSkillMilestones()
{
    // Check for skill milestones and achievements
    for (int i = 0; i < skillLevels.Length; i++)
    {
        if (skillLevels[i] >= 10 && skillLevels[i] % 10 == 0)
        {
            string skillName = GetSkillName(i);
            Debug.Log($"Skill Milestone: {skillName} reached level
{skillLevels[i]}");

            // Trigger milestone rewards
            OnSkillMilestone(i, skillLevels[i]);
        }
```

```
        }
    }

    void OnSkillMilestone(int skillIndex, int milestoneLevel)
    {
        // Grant milestone rewards
        float bonusExperience = milestoneLevel * 100f;
        skillExperience[skillIndex] += bonusExperience;

        // Show milestone notification
        string skillName = GetSkillName(skillIndex);
        ShowMilestoneNotification(skillName, milestoneLevel);
    }

    void ShowMilestoneNotification(string skillName, int milestone)
    {
        string message = $"fiÜfi¶fiçfiØfiÇfi∞ fiàfi®fiçfi™! {skillName} - fiàfi®fiçfi™
{milestone} fiàfi¶fiÅfi©fiÜfi™"; // "Skill achievement! [Skill] - Level [X] milestone"

        UISystem.Instance?.ShowCulturalNotification(message, 5.0f);
    }

    public SkillSnapshot GetSkillSnapshot()
    {
        float culturalCompetency = 0f;

        if (culturalSkills.Length > 0)
        {
            float totalCulturalLevel = 0f;
            for (int i = 0; i < culturalSkills.Length; i++)
            {
```

```
            totalCulturalLevel += culturalSkills[i].currentLevel;
        }
        culturalCompetency = totalCulturalLevel / culturalSkills.Length;
    }

    return new SkillSnapshot
    {
        total_skills = playerSkills.Length + culturalSkills.Length,
        average_skill_level = GetAverageSkillLevel(),
        cultural_competency = culturalCompetency,
        highest_skill = GetHighestSkill(),
        unlocked_skills = GetUnlockedSkillCount(),
        master_skills = GetMasterSkillCount()
    };
}

float GetAverageSkillLevel()
{
    if (skillLevels.Length == 0) return 0f;

    float total = 0f;
    for (int i = 0; i < skillLevels.Length; i++)
    {
        total += skillLevels[i];
    }
    return total / skillLevels.Length;
}

string GetHighestSkill()
{
    int highestLevel = 0;
```

```
    string highestSkill = "";

    for (int i = 0; i < skillLevels.Length; i++)
    {
        if (skillLevels[i] > highestLevel)
        {
            highestLevel = skillLevels[i];
            highestSkill = GetSkillName(i);
        }
    }

    return highestSkill;
}

int GetUnlockedSkillCount()
{
    int count = 0;
    for (int i = 0; i < skillUnlocked.Length; i++)
    {
        if (skillUnlocked[i]) count++;
    }
    return count;
}

int GetMasterSkillCount()
{
    int count = 0;
    for (int i = 0; i < skillLevels.Length; i++)
    {
        if (skillLevels[i] >= maxSkillLevel) count++;
    }
```

```
        return count;
    }

    [System.Serializable]
    public class SkillSnapshot
    {
        public int total_skills;
        public float average_skill_level;
        public float cultural_competency;
        public string highest_skill;
        public int unlocked_skills;
        public int master_skills;
    }
}
```

35. ParticleSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class ParticleSystem : MonoBehaviour
{
    [BurstCompile]
    struct MaldivianParticleSimulation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> positions;
        [ReadOnly] public NativeArray<float3> velocities;
```

```
    [ReadOnly] public NativeArray<float> lifetimes;
    [ReadOnly] public NativeArray<float4> colors;
    [WriteOnly] public NativeArray<float3> newPositions;
    [WriteOnly] public NativeArray<float> newLifetimes;
    [WriteOnly] public NativeArray<bool> activeParticles;

    public float deltaTime;
    public float3 gravity;
    public float drag;
    public float3 windForce;

    public void Execute(int index)
    {
        float3 position = positions[index];
        float3 velocity = velocities[index];
        float lifetime = lifetimes[index];

        // Apply physics simulation
        float3 newVelocity = velocity + (gravity + windForce) * deltaTime;
        newVelocity *= (1.0f - drag * deltaTime);

        float3 newPosition = position + newVelocity * deltaTime;
        float newLifetime = lifetime - deltaTime;

        newPositions[index] = newPosition;
        newLifetimes[index] = newLifetime;
        activeParticles[index] = newLifetime > 0.0f;
    }
}

[BurstCompile]
```

```
    struct CulturalParticleEffects : IJob
    {
        public NativeArray<float4> prayerParticleColors;
        public NativeArray<float3> oceanSprayPositions;
        public NativeArray<float4> traditionalSmokeColors;
        public NativeArray<float3> celebrationSparkPositions;

        public NativeArray<float4> finalParticleColors;
        public NativeArray<float3> finalParticlePositions;

        public void Execute()
        {
            ProcessPrayerParticles();
            ProcessOceanSpray();
            ProcessTraditionalSmoke();
            ProcessCelebrationSparks();
        }

        [BurstCompile]
        void ProcessPrayerParticles()
        {
            // Generate spiritual particle effects for prayer times
            for (int i = 0; i < prayerParticleColors.Length; i++)
            {
                float4 baseColor = new float4(0.9f, 0.8f, 0.6f, 0.7f); // Warm spiritual
glow
                float timeOffset = (float)i / prayerParticleColors.Length;
                float alpha = math.sin(timeOffset * math.PI * 2) * 0.3f + 0.4f;

                prayerParticleColors[i] = new float4(baseColor.xyz, alpha);
            }
```

```
    }

    [BurstCompile]
    void ProcessOceanSpray()
    {
        // Simulate realistic ocean spray patterns
        for (int i = 0; i < oceanSprayPositions.Length; i++)
        {
            float angle = (float)i / oceanSprayPositions.Length * math.PI * 2;
            float radius = 2.0f + math.sin(angle * 3) * 0.5f;

            oceanSprayPositions[i] = new float3(
                math.cos(angle) * radius,
                math.sin(i * 0.1f) * 0.5f,
                math.sin(angle) * radius * 0.3f
            );
        }
    }

    [BurstCompile]
    void ProcessTraditionalSmoke()
    {
        // Traditional hearth smoke colors (from cooking, incense)
        for (int i = 0; i < traditionalSmokeColors.Length; i++)
        {
            float4 smokeColor = new float4(0.4f, 0.3f, 0.2f, 0.6f); // Natural smoke
            float variation = math.sin((float)i * 0.5f) * 0.1f;

            traditionalSmokeColors[i] = smokeColor + new float4(variation,
variation, variation, 0);
        }
```

```
    }

    [BurstCompile]
    void ProcessCelebrationSparks()
    {
        // Festive spark effects for cultural celebrations
        for (int i = 0; i < celebrationSparkPositions.Length; i++)
        {
            float angle = (float)i / celebrationSparkPositions.Length * math.PI * 2;
            float height = (float)i / celebrationSparkPositions.Length * 5.0f;

            celebrationSparkPositions[i] = new float3(
                math.cos(angle) * (1.0f + height * 0.2f),
                height,
                math.sin(angle) * (1.0f + height * 0.2f)
            );
        }
    }
}

public static ParticleSystem Instance { get; private set; }

[Header("Maldivian Particle Effects")]
public bool enablePrayerParticles = true;
public bool enableOceanSpray = true;
public bool enableTraditionalSmoke = true;
public bool enableCelebrationSparks = true;

[Header("Performance Settings")]
public int maxParticles = 1000;
public bool useGPUInstancing = true;
```

```
    public bool enableLOD = true;

    [Header("Cultural Authenticity")]
    public bool respectReligiousContexts = true;
    public bool useNaturalColors = true;
    public bool simulateLocalWeather = true;

    private ParticleSystem[] particlePools;
    private int currentParticleIndex;

    void Awake()
    {
        Instance = this;
        InitializeParticleSystem();
    }

    void InitializeParticleSystem()
    {
        // Initialize particle pools
        particlePools = new ParticleSystem[10];

        int particleCount = 500;
        var positions = new NativeArray<float3>(particleCount, Allocator.TempJob);
        var velocities = new NativeArray<float3>(particleCount, Allocator.TempJob);
        var lifetimes = new NativeArray<float>(particleCount, Allocator.TempJob);
        var colors = new NativeArray<float4>(particleCount, Allocator.TempJob);
        var newPositions = new NativeArray<float3>(particleCount,
Allocator.TempJob);
        var newLifetimes = new NativeArray<float>(particleCount,
Allocator.TempJob);
```

```
        var activeParticles = new NativeArray<bool>(particleCount,
Allocator.TempJob);

        // Initialize particle data
        for (int i = 0; i < particleCount; i++)
        {
            positions[i] = UnityEngine.Random.insideUnitSphere * 10.0f;
            velocities[i] = UnityEngine.Random.insideUnitSphere * 2.0f;
            lifetimes[i] = UnityEngine.Random.Range(1.0f, 5.0f);
            colors[i] = new float4(UnityEngine.Random.ColorHSV(), 0.8f);
        }

        var simulationJob = new MaldivianParticleSimulation
        {
            positions = positions,
            velocities = velocities,
            lifetimes = lifetimes,
            colors = colors,
            newPositions = newPositions,
            newLifetimes = newLifetimes,
            activeParticles = activeParticles,
            deltaTime = Time.deltaTime,
            gravity = new float3(0, -9.81f, 0),
            drag = 0.1f,
            windForce = new float3(2.0f, 0, 1.0f)
        };

        // Cultural particle effects
        int prayerParticles = 100;
        int oceanSpray = 150;
        int traditionalSmoke = 80;
```

```
        int celebrationSparks = 120;

        var prayerColors = new NativeArray<float4>(prayerParticles,
Allocator.TempJob);
        var oceanPositions = new NativeArray<float3>(oceanSpray,
Allocator.TempJob);
        var smokeColors = new NativeArray<float4>(traditionalSmoke,
Allocator.TempJob);
        var sparkPositions = new NativeArray<float3>(celebrationSparks,
Allocator.TempJob);
        var finalColors = new NativeArray<float4>(prayerParticles +
traditionalSmoke, Allocator.TempJob);
        var finalPositions = new NativeArray<float3>(oceanSpray +
celebrationSparks, Allocator.TempJob);

        var culturalJob = new CulturalParticleEffects
        {
            prayerParticleColors = prayerColors,
            oceanSprayPositions = oceanPositions,
            traditionalSmokeColors = smokeColors,
            celebrationSparkPositions = sparkPositions,
            finalParticleColors = finalColors,
            finalParticlePositions = finalPositions
        };

        JobHandle simulationHandle = simulationJob.Schedule(particleCount, 32);
        JobHandle culturalHandle = culturalJob.Schedule(simulationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessParticleSimulation(newPositions, newLifetimes, activeParticles);
```

```
        ProcessCulturalEffects(finalColors, finalPositions);

        // Cleanup
        positions.Dispose();
        velocities.Dispose();
        lifetimes.Dispose();
        colors.Dispose();
        newPositions.Dispose();
        newLifetimes.Dispose();
        activeParticles.Dispose();
        prayerColors.Dispose();
        oceanPositions.Dispose();
        smokeColors.Dispose();
        sparkPositions.Dispose();
        finalColors.Dispose();
        finalPositions.Dispose();

        StartParticleSystems();
    }

    void ProcessParticleSimulation(NativeArray<float3> positions,
NativeArray<float> lifetimes, NativeArray<bool> active)
    {
        // Update particle positions and lifetimes
        int activeCount = 0;
        for (int i = 0; i < active.Length; i++)
        {
            if (active[i]) activeCount++;
        }

        Debug.Log($"Active particles: {activeCount}/{active.Length}");
```

```
    }

    void ProcessCulturalEffects(NativeArray<float4> colors, NativeArray<float3>
positions)
    {
        // Apply culturally appropriate particle effects
        Debug.Log($"Cultural particle effects processed: {colors.Length} colors,
{positions.Length} positions");
    }

    void StartParticleSystems()
    {
        // Initialize various particle systems
        if (enablePrayerParticles) CreatePrayerParticleSystem();
        if (enableOceanSpray) CreateOceanSpraySystem();
        if (enableTraditionalSmoke) CreateTraditionalSmokeSystem();
        if (enableCelebrationSparks) CreateCelebrationSparkSystem();
    }

    void CreatePrayerParticleSystem()
    {
        // Create spiritual particle effects for prayer times
        GameObject prayerSystem = new GameObject("PrayerParticles");
        prayerSystem.transform.SetParent(transform);

        var ps = prayerSystem.AddComponent<ParticleSystem>();
        var main = ps.main;
        main.startColor = new Color(0.9f, 0.8f, 0.6f, 0.7f);
        main.startLifetime = 3.0f;
        main.startSpeed = 1.0f;
        main.maxParticles = 50;
```

```
    var emission = ps.emission;
    emission.rateOverTime = 10;

    var shape = ps.shape;
    shape.shapeType = ParticleSystemShapeType.Circle;
    shape.radius = 2.0f;
}

void CreateOceanSpraySystem()
{
    // Create realistic ocean spray effects
    GameObject oceanSystem = new GameObject("OceanSpray");
    oceanSystem.transform.SetParent(transform);

    var ps = oceanSystem.AddComponent<ParticleSystem>();
    var main = ps.main;
    main.startColor = new Color(0.8f, 0.9f, 1.0f, 0.6f);
    main.startLifetime = 2.0f;
    main.startSpeed = 5.0f;
    main.maxParticles = 200;

    var velocityOverLifetime = ps.velocityOverLifetime;
    velocityOverLifetime.enabled = true;
    velocityOverLifetime.space = ParticleSystemSimulationSpace.World;
    velocityOverLifetime.y = new ParticleSystem.MinMaxCurve(-2.0f);
}

void CreateTraditionalSmokeSystem()
{
    // Create traditional hearth smoke effects
```

```
    GameObject smokeSystem = new GameObject("TraditionalSmoke");
    smokeSystem.transform.SetParent(transform);

    var ps = smokeSystem.AddComponent<ParticleSystem>();
    var main = ps.main;
    main.startColor = new Color(0.4f, 0.3f, 0.2f, 0.5f);
    main.startLifetime = 4.0f;
    main.startSpeed = 0.5f;
    main.maxParticles = 80;
    main.startSize = 0.3f;

    var noise = ps.noise;
    noise.enabled = true;
    noise.frequency = 0.5f;
    noise.strength = 1.0f;
}

void CreateCelebrationSparkSystem()
{
    // Create festive spark effects
    GameObject sparkSystem = new GameObject("CelebrationSparks");
    sparkSystem.transform.SetParent(transform);

    var ps = sparkSystem.AddComponent<ParticleSystem>();
    var main = ps.main;
    main.startColor = new Color(1.0f, 0.8f, 0.2f, 1.0f);
    main.startLifetime = 1.5f;
    main.startSpeed = 8.0f;
    main.maxParticles = 100;
    main.startSize = 0.1f;
```

```
        var trails = ps.trails;
        trails.enabled = true;
        trails.lifetime = 0.5f;
        trails.minimumVertexDistance = 0.1f;
    }

    public void TriggerPrayerEffect(Vector3 position)
    {
        if (!enablePrayerParticles || !respectReligiousContexts) return;

        // Create spiritual particle burst at prayer time
        CreateParticleBurst(position, 30, new Color(0.9f, 0.8f, 0.6f, 0.8f), 2.0f);
    }

    public void TriggerOceanSpray(Vector3 position, float intensity)
    {
        if (!enableOceanSpray) return;

        // Create ocean spray at coastline
        CreateParticleBurst(position, (int)(50 * intensity), new Color(0.8f, 0.9f, 1.0f,
0.7f), 3.0f);
    }

    public void TriggerTraditionalSmoke(Vector3 position)
    {
        if (!enableTraditionalSmoke) return;

        // Create traditional hearth smoke
        CreateParticleBurst(position, 20, new Color(0.4f, 0.3f, 0.2f, 0.6f), 4.0f);
    }
```

```
public void TriggerCelebrationSparks(Vector3 position)
{
    if (!enableCelebrationSparks) return;

    // Create festive sparks for celebrations
    CreateFireworks(position, 15);
}

void CreateParticleBurst(Vector3 position, int count, Color color, float lifetime)
{
    // Implementation for particle burst
    Debug.Log($"Particle burst: {count} particles at {position}");
}

void CreateFireworks(Vector3 position, int count)
{
    // Implementation for fireworks effect
    Debug.Log($"Fireworks: {count} sparks at {position}");
}

public ParticleSnapshot GetParticleSnapshot()
{
    return new ParticleSnapshot
    {
        active_systems = GetActiveSystemCount(),
        total_particles = GetTotalParticleCount(),
        cultural_effects_active = GetCulturalEffectCount(),
        performance_rating = GetPerformanceRating()
    };
}
```

```
int GetActiveSystemCount()
{
    int count = 0;
    if (enablePrayerParticles) count++;
    if (enableOceanSpray) count++;
    if (enableTraditionalSmoke) count++;
    if (enableCelebrationSparks) count++;
    return count;
}

int GetTotalParticleCount()
{
    // Calculate total particles across all systems
    return 430; // Simplified for this implementation
}

int GetCulturalEffectCount()
{
    int count = 0;
    if (enablePrayerParticles) count++;
    if (enableTraditionalSmoke) count++;
    if (enableCelebrationSparks) count++;
    return count;
}

float GetPerformanceRating()
{
    // Calculate performance based on active particles
    int totalParticles = GetTotalParticleCount();
    return Mathf.Clamp01(1.0f - (totalParticles / (float)maxParticles));
}
```

```
    [System.Serializable]
    public class ParticleSnapshot
    {
        public int active_systems;
        public int total_particles;
        public int cultural_effects_active;
        public float performance_rating;
    }
}
```

36. ShadowSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using UnityEngine.Rendering;

[BurstCompile]
public class ShadowSystem : MonoBehaviour
{
    [BurstCompile]
    struct OptimizedShadowCalculation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> lightPositions;
        [ReadOnly] public NativeArray<float3> objectPositions;
        [ReadOnly] public NativeArray<float3> objectBounds;
        [WriteOnly] public NativeArray<float4> shadowData;
```

```
    [WriteOnly] public NativeArray<bool> castShadows;
    [WriteOnly] public NativeArray<int> shadowLOD;

    public float3 mainLightDirection;
    public float shadowDistance;
    public int shadowQuality;
    public float performanceTarget;

    public void Execute(int index)
    {
        float3 objectPos = objectPositions[index];
        float3 objectBound = objectBounds[index];

        // Calculate shadow casting eligibility
        bool shouldCastShadow = ShouldCastShadow(objectPos, objectBound);
        int lodLevel = CalculateShadowLOD(objectPos);
        float4 shadowInfo = CalculateShadowData(objectPos, objectBound,
lodLevel);

        castShadows[index] = shouldCastShadow;
        shadowLOD[index] = lodLevel;
        shadowData[index] = shadowInfo;
    }

    [BurstCompile]
    bool ShouldCastShadow(float3 position, float3 bounds)
    {
        // Distance-based shadow culling
        float distance = math.length(position);
        if (distance > shadowDistance) return false;
```

```
    // Size-based culling (small objects at distance)
    float maxBound = math.max(math.max(bounds.x, bounds.y), bounds.z);
    if (distance > shadowDistance * 0.5f && maxBound < 1.0f) return false;

    return true;
}

[BurstCompile]
int CalculateShadowLOD(float3 position)
{
    float distance = math.length(position);
    float distanceRatio = distance / shadowDistance;

    // LOD levels: 0=High, 1=Medium, 2=Low
    if (distanceRatio < 0.3f) return 0;
    else if (distanceRatio < 0.7f) return 1;
    else return 2;
}

[BurstCompile]
float4 CalculateShadowData(float3 position, float3 bounds, int lod)
{
    // Calculate shadow map resolution and bias based on LOD
    int resolution = shadowQuality switch
    {
        0 => 2048 >> lod, // Ultra
        1 => 1024 >> lod, // High
        2 => 512 >> lod,  // Medium
        _ => 256 >> lod   // Low
    };
```

```
        float bias = 0.1f * (lod + 1);
        float normalBias = 0.05f * (lod + 1);

        return new float4(resolution, bias, normalBias, lod);
    }
}

[BurstCompile]
struct MaldivianShadowOptimization : IJob
{
    public NativeArray<float3> palmTreePositions;
    public NativeArray<float3> buildingPositions;
    public NativeArray<float3> boatPositions;
    public NativeArray<float3> terrainFeatures;

    public NativeArray<float4> optimizedShadowDistances;
    public NativeArray<bool> culturalShadowFlags;

    public float timeOfDay;
    public float prayerTimeWeight;

    public void Execute()
    {
        OptimizePalmTreeShadows();
        OptimizeBuildingShadows();
        OptimizeBoatShadows();
        OptimizeTerrainShadows();
        ApplyCulturalShadowRules();
    }

    [BurstCompile]
```

```
    void OptimizePalmTreeShadows()
    {
        // Palm trees need detailed shadows but can be optimized
        for (int i = 0; i < palmTreePositions.Length; i++)
        {
            float3 pos = palmTreePositions[i];
            float distance = math.length(pos);

            // Palm fronds create complex shadows - adjust quality based on
distance
            float shadowDistance = math.max(50.0f, 100.0f - distance * 0.5f);
            optimizedShadowDistances[i] = new float4(shadowDistance, 1.0f, 0.8f,
1.0f);
        }
    }

    [BurstCompile]
    void OptimizeBuildingShadows()
    {
        // Traditional buildings need accurate shadows
        for (int i = 0; i < buildingPositions.Length; i++)
        {
            float3 pos = buildingPositions[i];
            float distance = math.length(pos);

            // Important for cultural authenticity - keep higher quality
            float shadowDistance = math.max(75.0f, 150.0f - distance * 0.3f);
            optimizedShadowDistances[palmTreePositions.Length + i] = new
float4(shadowDistance, 1.0f, 0.9f, 1.0f);
        }
    }
```

```
    [BurstCompile]
    void OptimizeBoatShadows()
    {
        // Boat shadows on water - optimize for performance
        for (int i = 0; i < boatPositions.Length; i++)
        {
            float3 pos = boatPositions[i];
            float distance = math.length(pos);

            // Boats on water can use lower shadow quality
            float shadowDistance = math.max(30.0f, 80.0f - distance * 0.4f);
            optimizedShadowDistances[palmTreePositions.Length +
buildingPositions.Length + i] = new float4(shadowDistance, 0.7f, 0.5f, 0.8f);
        }
    }

    [BurstCompile]
    void OptimizeTerrainShadows()
    {
        // Terrain and vegetation shadows
        for (int i = 0; i < terrainFeatures.Length; i++)
        {
            float3 pos = terrainFeatures[i];
            float distance = math.length(pos);

            // Terrain shadows can be optimized aggressively
            float shadowDistance = math.max(40.0f, 100.0f - distance * 0.6f);
            optimizedShadowDistances[palmTreePositions.Length +
buildingPositions.Length + boatPositions.Length + i] = new
float4(shadowDistance, 0.6f, 0.4f, 0.7f);
```

```
            }
        }

        [BurstCompile]
        void ApplyCulturalShadowRules()
        {
            // Apply special shadow rules during prayer times
            bool isPrayerTime = prayerTimeWeight > 0.7f;

            if (isPrayerTime)
            {
                // During prayer times, enhance shadow quality for religious buildings
                for (int i = palmTreePositions.Length; i < palmTreePositions.Length +
buildingPositions.Length; i++)
                {
                    culturalShadowFlags[i] = true; // Enhanced shadow quality
                }
            }
        }
    }

    public static ShadowSystem Instance { get; private set; }

    [Header("Shadow Quality Settings")]
    public ShadowQuality shadowQuality = ShadowQuality.High;
    public float shadowDistance = 100.0f;
    public bool enableDynamicShadows = true;
    public bool enableCulturalShadowOptimization = true;

    [Header("Performance Optimization")]
    public bool enableShadowLOD = true;
```

```
    public bool enableShadowCaching = true;
    public float performanceTarget = 60.0f;
    public bool adaptiveShadowDistance = true;

    [Header("Maldivian Cultural Shadows")]
    public bool respectPrayerTimeShadows = true;
    public bool enhanceTraditionalArchitecture = true;
    public bool optimizeOceanShadows = true;
    public bool preserveCulturalShadowAccuracy = true;

    private Light mainLight;
    private UnityEngine.Rendering.Universal.UniversalAdditionalCameraData
cameraData;
    private float currentShadowDistance;
    private float shadowPerformance;

    public enum ShadowQuality
    {
        Low = 0,
        Medium = 1,
        High = 2,
        Ultra = 3
    }

    void Awake()
    {
        Instance = this;
        InitializeShadowSystem();
    }

    void InitializeShadowSystem()
```

```
    {
        // Get main light and camera
        mainLight = FindObjectOfType<Light>();
        Camera mainCamera = FindObjectOfType<Camera>();
        if (mainCamera != null)
        {
            cameraData =
mainCamera.GetComponent<UnityEngine.Rendering.Universal.UniversalAdditio
nalCameraData>();
        }

        currentShadowDistance = shadowDistance;

        // Initialize shadow calculations
        int objectCount = 200;
        var lightPositions = new NativeArray<float3>(4, Allocator.TempJob);
        var objectPositions = new NativeArray<float3>(objectCount,
Allocator.TempJob);
        var objectBounds = new NativeArray<float3>(objectCount,
Allocator.TempJob);
        var shadowData = new NativeArray<float4>(objectCount,
Allocator.TempJob);
        var castShadows = new NativeArray<bool>(objectCount,
Allocator.TempJob);
        var shadowLOD = new NativeArray<int>(objectCount, Allocator.TempJob);

        // Initialize with sample data
        lightPositions[0] = new float3(0, 10, 0); // Main sun light
        for (int i = 0; i < objectCount; i++)
        {
            objectPositions[i] = UnityEngine.Random.insideUnitSphere * 50.0f;
```

```
            objectBounds[i] = UnityEngine.Random.Range(0.5f, 3.0f) * Vector3.one;
        }

        var shadowJob = new OptimizedShadowCalculation
        {
            lightPositions = lightPositions,
            objectPositions = objectPositions,
            objectBounds = objectBounds,
            shadowData = shadowData,
            castShadows = castShadows,
            shadowLOD = shadowLOD,
            mainLightDirection = mainLight ? mainLight.transform.forward :
Vector3.down,
            shadowDistance = shadowDistance,
            shadowQuality = (int)shadowQuality,
            performanceTarget = performanceTarget
        };

        // Maldivian cultural shadow optimization
        int palmTrees = 30;
        int buildings = 25;
        int boats = 15;
        int terrain = 50;

        var palmPositions = new NativeArray<float3>(palmTrees,
Allocator.TempJob);
        var buildingPositions = new NativeArray<float3>(buildings,
Allocator.TempJob);
        var boatPositions = new NativeArray<float3>(boats, Allocator.TempJob);
        var terrainPositions = new NativeArray<float3>(terrain, Allocator.TempJob);
```

```
    var optimizedDistances = new NativeArray<float4>(palmTrees + buildings +
boats + terrain, Allocator.TempJob);
    var culturalFlags = new NativeArray<bool>(palmTrees + buildings + boats +
terrain, Allocator.TempJob);

    // Initialize Maldivian scene data
    for (int i = 0; i < palmTrees; i++)
    {
        palmPositions[i] = new float3(UnityEngine.Random.Range(-20, 20), 0,
UnityEngine.Random.Range(-20, 20));
    }
    for (int i = 0; i < buildings; i++)
    {
        buildingPositions[i] = new float3(UnityEngine.Random.Range(-30, 30), 0,
UnityEngine.Random.Range(-30, 30));
    }
    for (int i = 0; i < boats; i++)
    {
        boatPositions[i] = new float3(UnityEngine.Random.Range(-40, 40), 0,
UnityEngine.Random.Range(-40, 40));
    }
    for (int i = 0; i < terrain; i++)
    {
        terrainPositions[i] = new float3(UnityEngine.Random.Range(-50, 50), 0,
UnityEngine.Random.Range(-50, 50));
    }

    var culturalJob = new MaldivianShadowOptimization
    {
        palmTreePositions = palmPositions,
        buildingPositions = buildingPositions,
```

```
    boatPositions = boatPositions,
    terrainFeatures = terrainPositions,
    optimizedShadowDistances = optimizedDistances,
    culturalShadowFlags = culturalFlags,
    timeOfDay = Time.time / 86400.0f, // Convert to day fraction
    prayerTimeWeight = GetPrayerTimeWeight()
};

JobHandle shadowHandle = shadowJob.Schedule(objectCount, 16);
JobHandle culturalHandle = culturalJob.Schedule(shadowHandle);
culturalHandle.Complete();

// Process results
ProcessShadowCalculations(shadowData, castShadows, shadowLOD);
ProcessCulturalOptimizations(optimizedDistances, culturalFlags);

// Cleanup
lightPositions.Dispose();
objectPositions.Dispose();
objectBounds.Dispose();
shadowData.Dispose();
castShadows.Dispose();
shadowLOD.Dispose();
palmPositions.Dispose();
buildingPositions.Dispose();
boatPositions.Dispose();
terrainPositions.Dispose();
optimizedDistances.Dispose();
culturalFlags.Dispose();

StartShadowMonitoring();
```

```
    }

    void ProcessShadowCalculations(NativeArray<float4> data,
NativeArray<bool> castShadows, NativeArray<int> lod)
    {
        int shadowCasters = 0;
        int totalLOD = 0;

        for (int i = 0; i < castShadows.Length; i++)
        {
            if (castShadows[i]) shadowCasters++;
            totalLOD += lod[i];
        }

        float averageLOD = (float)totalLOD / castShadows.Length;
        Debug.Log($"Shadow calculation complete: {shadowCasters} shadow
casters, average LOD: {averageLOD:F2}");
    }

    void ProcessCulturalOptimizations(NativeArray<float4> distances,
NativeArray<bool> culturalFlags)
    {
        int culturalOptimizations = 0;
        for (int i = 0; i < culturalFlags.Length; i++)
        {
            if (culturalFlags[i]) culturalOptimizations++;
        }

        Debug.Log($"Cultural shadow optimizations applied: {culturalOptimizations}
objects");
    }
```

```
    void StartShadowMonitoring()
    {
        // Monitor shadow performance
        InvokeRepeating("UpdateShadowPerformance", 1.0f, 2.0f);
        InvokeRepeating("AdjustShadowSettings", 5.0f, 10.0f);
    }

    void UpdateShadowPerformance()
    {
        // Calculate current shadow performance
        shadowPerformance = CalculateShadowPerformance();

        if (adaptiveShadowDistance)
        {
            // Adjust shadow distance based on performance
            if (shadowPerformance < performanceTarget * 0.8f)
            {
                currentShadowDistance = Mathf.Max(50.0f, currentShadowDistance -
10.0f);
            }
            else if (shadowPerformance > performanceTarget * 1.1f)
            {
                currentShadowDistance = Mathf.Min(shadowDistance,
currentShadowDistance + 5.0f);
            }
        }
    }

    float CalculateShadowPerformance()
    {
```

```
        // Simplified performance calculation
        return performanceTarget * 0.95f; // Placeholder
    }

    void AdjustShadowSettings()
    {
        if (!adaptiveShadowDistance) return;

        // Apply adaptive shadow settings
        if (mainLight != null)
        {
            mainLight.shadowStrength = Mathf.Lerp(0.5f, 1.0f, shadowPerformance /
performanceTarget);
            mainLight.shadowBias = Mathf.Lerp(0.2f, 0.05f, shadowPerformance /
performanceTarget);
            mainLight.shadowNormalBias = Mathf.Lerp(0.4f, 0.1f,
shadowPerformance / performanceTarget);
        }

        if (cameraData != null)
        {
            cameraData.renderShadows = shadowPerformance >
performanceTarget * 0.7f;
        }
    }

    float GetPrayerTimeWeight()
    {
        // Check if it's near prayer time
        if (PrayerTimeSystem.Instance != null)
        {
```

```
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0.0f;
}

public void SetShadowQuality(ShadowQuality quality)
{
    shadowQuality = quality;

    if (mainLight != null)
    {
        mainLight.shadowResolution = quality switch
        {
            ShadowQuality.Low => LightShadowResolution.Low,
            ShadowQuality.Medium => LightShadowResolution.Medium,
            ShadowQuality.High => LightShadowResolution.High,
            ShadowQuality.Ultra => LightShadowResolution.VeryHigh,
            _ => LightShadowResolution.High
        };
    }
}

public ShadowSnapshot GetShadowSnapshot()
{
    return new ShadowSnapshot
    {
        shadow_quality = shadowQuality.ToString(),
        shadow_distance = currentShadowDistance,
        performance_rating = shadowPerformance,
        active_shadows = CountActiveShadows(),
        cultural_optimizations = enableCulturalShadowOptimization,
```

```
            adaptive_performance = adaptiveShadowDistance
        };
    }

    int CountActiveShadows()
    {
        // Count currently active shadow casters
        int count = 0;
        GameObject[] allObjects = FindObjectsOfType<GameObject>();
        foreach (GameObject obj in allObjects)
        {
            Renderer renderer = obj.GetComponent<Renderer>();
            if (renderer != null && renderer.shadowCastingMode !=
ShadowCastingMode.Off)
            {
                count++;
            }
        }
        return count;
    }

    [System.Serializable]
    public class ShadowSnapshot
    {
        public string shadow_quality;
        public float shadow_distance;
        public float performance_rating;
        public int active_shadows;
        public bool cultural_optimizations;
        public bool adaptive_performance;
    }
```

```
}
```

37. WildlifeSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class WildlifeSystem : MonoBehaviour
{
    [BurstCompile]
    struct MarineLifeSimulation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> fishPositions;
        [ReadOnly] public NativeArray<float3> fishVelocities;
        [ReadOnly] public NativeArray<float> fishSizes;
        [ReadOnly] public NativeArray<int> fishSpecies;
        [WriteOnly] public NativeArray<float3> newPositions;
        [WriteOnly] public NativeArray<float3> newVelocities;
        [WriteOnly] public NativeArray<bool> activeFish;

        public float deltaTime;
        public float3 currentDirection;
        public float3 avoidanceCenter;
        public float separationDistance;
        public float alignmentDistance;
```

```
    public float cohesionDistance;

    public void Execute(int index)
    {
        float3 position = fishPositions[index];
        float3 velocity = fishVelocities[index];
        float size = fishSizes[index];
        int species = fishSpecies[index];

        // Apply flocking behavior
        float3 separation = CalculateSeparation(index, position);
        float3 alignment = CalculateAlignment(index, velocity);
        float3 cohesion = CalculateCohesion(index, position);
        float3 avoidance = CalculateAvoidance(position, avoidanceCenter);

        // Combine behaviors with species-specific weights
        float3 newVelocity = velocity +
            (separation * GetSeparationWeight(species)) +
            (alignment * GetAlignmentWeight(species)) +
            (cohesion * GetCohesionWeight(species)) +
            (avoidance * 2.0f) +
            (currentDirection * 0.3f);

        // Limit velocity based on species
        float maxSpeed = GetMaxSpeed(species) * size;
        newVelocity = math.normalize(newVelocity) *
math.min(math.length(newVelocity), maxSpeed);

        float3 newPosition = position + newVelocity * deltaTime;
        bool active = IsInBounds(newPosition);
```

```
    newPositions[index] = newPosition;
    newVelocities[index] = newVelocity;
    activeFish[index] = active;
}

[BurstCompile]
float3 CalculateSeparation(int index, float3 position)
{
    float3 steer = float3.zero;
    int count = 0;

    for (int i = 0; i < fishPositions.Length; i++)
    {
        if (i == index) continue;

        float distance = math.length(fishPositions[i] - position);
        if (distance > 0 && distance < separationDistance)
        {
            float3 diff = position - fishPositions[i];
            diff = math.normalize(diff);
            diff /= distance; // Weight by distance
            steer += diff;
            count++;
        }
    }

    if (count > 0)
    {
        steer /= count;
        steer = math.normalize(steer);
    }
```

```
    return steer;
}

[BurstCompile]
float3 CalculateAlignment(int index, float3 velocity)
{
    float3 sum = float3.zero;
    int count = 0;

    for (int i = 0; i < fishVelocities.Length; i++)
    {
        if (i == index) continue;

        float distance = math.length(fishPositions[i] - fishPositions[index]);
        if (distance > 0 && distance < alignmentDistance)
        {
            sum += fishVelocities[i];
            count++;
        }
    }

    if (count > 0)
    {
        sum /= count;
        sum = math.normalize(sum);
    }

    return sum;
}
```

```
[BurstCompile]
float3 CalculateCohesion(int index, float3 position)
{
    float3 sum = float3.zero;
    int count = 0;

    for (int i = 0; i < fishPositions.Length; i++)
    {
        if (i == index) continue;

        float distance = math.length(fishPositions[i] - position);
        if (distance > 0 && distance < cohesionDistance)
        {
            sum += fishPositions[i];
            count++;
        }
    }

    if (count > 0)
    {
        sum /= count;
        return math.normalize(sum - position);
    }

    return float3.zero;
}

[BurstCompile]
float3 CalculateAvoidance(float3 position, float3 avoidCenter)
{
    float distance = math.length(avoidCenter - position);
```

```
    if (distance < 5.0f)
    {
        float3 avoid = position - avoidCenter;
        return math.normalize(avoid) * (1.0f - distance / 5.0f);
    }
    return float3.zero;
}

[BurstCompile]
bool IsInBounds(float3 position)
{
    // Keep fish within ocean bounds
    return position.y >= -20.0f && position.y <= 0.0f &&
           math.length(position.xz) < 200.0f;
}

[BurstCompile]
float GetSeparationWeight(int species)
{
    return species switch
    {
        0 => 1.5f, // Tuna
        1 => 1.2f, // Reef fish
        2 => 2.0f, // Small schooling fish
        3 => 0.8f, // Sharks
        _ => 1.0f
    };
}

[BurstCompile]
float GetAlignmentWeight(int species)
```

```
{
    return species switch
    {
        0 => 1.0f, // Tuna
        1 => 0.6f, // Reef fish
        2 => 1.8f, // Small schooling fish
        3 => 0.3f, // Sharks
        _ => 1.0f
    };
}

[BurstCompile]
float GetCohesionWeight(int species)
{
    return species switch
    {
        0 => 1.0f, // Tuna
        1 => 0.7f, // Reef fish
        2 => 1.6f, // Small schooling fish
        3 => 0.4f, // Sharks
        _ => 1.0f
    };
}

[BurstCompile]
float GetMaxSpeed(int species)
{
    return species switch
    {
        0 => 8.0f,  // Tuna
        1 => 4.0f,  // Reef fish
```

```
            2 => 6.0f,  // Small schooling fish
            3 => 10.0f, // Sharks
            _ => 5.0f
        };
    }
}

[BurstCompile]
struct MaldivianMarineEcosystem : IJob
{
    public NativeArray<float> tunaPopulation;
    public NativeArray<float> reefFishPopulation;
    public NativeArray<float> sharkPopulation;
    public NativeArray<float> turtlePopulation;
    public NativeArray<float> rayPopulation;

    public NativeArray<float> ecosystemHealth;
    public NativeArray<float> biodiversityIndex;
    public NativeArray<float> foodChainBalance;

    public float timeOfDay;
    public float seasonalFactor;
    public float humanImpact;

    public void Execute()
    {
        CalculateEcosystemHealth();
        CalculateBiodiversity();
        CalculateFoodChainBalance();
        ApplyMaldivianFactors();
    }
```

```
    [BurstCompile]
    void CalculateEcosystemHealth()
    {
        // Calculate overall ecosystem health
        float tunaHealth = tunaPopulation[0] / 1000.0f; // Target: 1000 tuna
        float reefHealth = reefFishPopulation[0] / 500.0f;   // Target: 500 reef fish
        float sharkHealth = sharkPopulation[0] / 50.0f;    // Target: 50 sharks
        float turtleHealth = turtlePopulation[0] / 100.0f;   // Target: 100 turtles
        float rayHealth = rayPopulation[0] / 75.0f;      // Target: 75 rays

        float avgHealth = (tunaHealth + reefHealth + sharkHealth + turtleHealth +
rayHealth) / 5.0f;
        ecosystemHealth[0] = math.saturate(avgHealth);
    }

    [BurstCompile]
    void CalculateBiodiversity()
    {
        // Calculate biodiversity index (Shannon diversity)
        float total = tunaPopulation[0] + reefFishPopulation[0] +
sharkPopulation[0] +
                      turtlePopulation[0] + rayPopulation[0];

        if (total > 0)
        {
            float p1 = tunaPopulation[0] / total;
            float p2 = reefFishPopulation[0] / total;
            float p3 = sharkPopulation[0] / total;
            float p4 = turtlePopulation[0] / total;
            float p5 = rayPopulation[0] / total;
```

```
            float diversity = 0.0f;
            if (p1 > 0) diversity -= p1 * math.log2(p1);
            if (p2 > 0) diversity -= p2 * math.log2(p2);
            if (p3 > 0) diversity -= p3 * math.log2(p3);
            if (p4 > 0) diversity -= p4 * math.log2(p4);
            if (p5 > 0) diversity -= p5 * math.log2(p5);

            biodiversityIndex[0] = math.saturate(diversity / 2.32f); // Max diversity
for 5 species
        }
    }

    [BurstCompile]
    void CalculateFoodChainBalance()
    {
        // Check predator-prey balance
        float tunaToSharkRatio = tunaPopulation[0] / math.max(1,
sharkPopulation[0]);
        float reefToSharkRatio = reefFishPopulation[0] / math.max(1,
sharkPopulation[0]);

        // Ideal ratios (prey to predator)
        float idealTunaSharkRatio = 20.0f;
        float idealReefSharkRatio = 10.0f;

        float balance = 1.0f - (math.abs(tunaToSharkRatio -
idealTunaSharkRatio) / idealTunaSharkRatio +
                                math.abs(reefToSharkRatio - idealReefSharkRatio) /
idealReefSharkRatio) * 0.5f;
```

```
            foodChainBalance[0] = math.saturate(balance);
        }

        [BurstCompile]
        void ApplyMaldivianFactors()
        {
            // Apply time-based factors
            float dawnFactor = math.saturate(math.sin(timeOfDay * math.PI * 2) *
2.0f);
            float duskFactor = math.saturate(math.sin((timeOfDay + 0.5f) * math.PI *
2) * 2.0f);

            // Fish are more active during dawn and dusk
            float activityFactor = math.max(dawnFactor, duskFactor);

            // Apply seasonal migration patterns
            float migrationFactor = math.sin(seasonalFactor * math.PI * 2) * 0.3f +
0.7f;

            // Apply human impact (fishing, tourism)
            float impactFactor = math.saturate(1.0f - humanImpact * 0.5f);

            // Adjust populations based on factors
            ecosystemHealth[0] *= activityFactor * migrationFactor * impactFactor;
            biodiversityIndex[0] *= impactFactor;
            foodChainBalance[0] *= migrationFactor;
        }
    }

    public static WildlifeSystem Instance { get; private set; }
```

```
[Header("Marine Life Settings")]
public int maxFishCount = 1000;
public bool enableFishAI = true;
public bool enableEcosystemSimulation = true;
public float ecosystemUpdateInterval = 5.0f;

[Header("Maldivian Species")]
public bool enableTuna = true;
public bool enableReefFish = true;
public bool enableSharks = true;
public bool enableTurtles = true;
public bool enableRays = true;

[Header("Cultural Respect")]
public bool respectMarineSanctuaries = true;
public bool simulateTraditionalFishing = true;
public bool preserveEndangeredSpecies = true;
public bool educationalWildlifeMode = true;

private GameObject[] fishObjects;
private NativeArray<float3> fishPositions;
private NativeArray<float3> fishVelocities;
private NativeArray<float> fishSizes;
private NativeArray<int> fishSpecies;
private float ecosystemHealth;
private float biodiversityIndex;
private float foodChainBalance;

public enum MarineSpecies
{
    YellowfinTuna,
```

```
        SkipjackTuna,
        ReefFish,
        ParrotFish,
        AngelFish,
        BlacktipReefShark,
        WhitetipReefShark,
        GreenSeaTurtle,
        HawksbillTurtle,
        MantaRay,
        EagleRay
    }

    void Awake()
    {
        Instance = this;
        InitializeWildlifeSystem();
    }

    void InitializeWildlifeSystem()
    {
        // Initialize fish arrays
        fishObjects = new GameObject[maxFishCount];
        fishPositions = new NativeArray<float3>(maxFishCount,
Allocator.Persistent);
        fishVelocities = new NativeArray<float3>(maxFishCount,
Allocator.Persistent);
        fishSizes = new NativeArray<float>(maxFishCount, Allocator.Persistent);
        fishSpecies = new NativeArray<int>(maxFishCount, Allocator.Persistent);

        // Initialize marine life simulation
        int fishCount = maxFishCount;
```

```
        var positions = new NativeArray<float3>(fishCount, Allocator.TempJob);
        var velocities = new NativeArray<float3>(fishCount, Allocator.TempJob);
        var sizes = new NativeArray<float>(fishCount, Allocator.TempJob);
        var species = new NativeArray<int>(fishCount, Allocator.TempJob);
        var newPositions = new NativeArray<float3>(fishCount, Allocator.TempJob);
        var newVelocities = new NativeArray<float3>(fishCount,
Allocator.TempJob);
        var activeFish = new NativeArray<bool>(fishCount, Allocator.TempJob);

        // Initialize fish data with Maldivian species distribution
        for (int i = 0; i < fishCount; i++)
        {
            positions[i] = new float3(
                UnityEngine.Random.Range(-100, 100),
                UnityEngine.Random.Range(-15, -2),
                UnityEngine.Random.Range(-100, 100)
            );
            velocities[i] = UnityEngine.Random.insideUnitSphere * 2.0f;
            sizes[i] = UnityEngine.Random.Range(0.3f, 2.5f);
            species[i] = GetRandomMaldivianSpecies();
        }

        var simulationJob = new MarineLifeSimulation
        {
            fishPositions = positions,
            fishVelocities = velocities,
            fishSizes = sizes,
            fishSpecies = species,
            newPositions = newPositions,
            newVelocities = newVelocities,
            activeFish = activeFish,
```

```
    deltaTime = Time.deltaTime,
    currentDirection = new float3(1, 0, 0.5f),
    avoidanceCenter = float3.zero,
    separationDistance = 3.0f,
    alignmentDistance = 5.0f,
    cohesionDistance = 10.0f
};

// Initialize ecosystem simulation
var tunaPop = new NativeArray<float>(1, Allocator.TempJob);
var reefPop = new NativeArray<float>(1, Allocator.TempJob);
var sharkPop = new NativeArray<float>(1, Allocator.TempJob);
var turtlePop = new NativeArray<float>(1, Allocator.TempJob);
var rayPop = new NativeArray<float>(1, Allocator.TempJob);
var health = new NativeArray<float>(1, Allocator.TempJob);
var diversity = new NativeArray<float>(1, Allocator.TempJob);
var balance = new NativeArray<float>(1, Allocator.TempJob);

// Set initial populations based on Maldivian marine ecosystem
tunaPop[0] = 300.0f;
reefPop[0] = 400.0f;
sharkPop[0] = 25.0f;
turtlePop[0] = 60.0f;
rayPop[0] = 40.0f;

var ecosystemJob = new MaldivianMarineEcosystem
{
    tunaPopulation = tunaPop,
    reefFishPopulation = reefPop,
    sharkPopulation = sharkPop,
    turtlePopulation = turtlePop,
```

```
    rayPopulation = rayPop,
    ecosystemHealth = health,
    biodiversityIndex = diversity,
    foodChainBalance = balance,
    timeOfDay = Time.time / 86400.0f,
    seasonalFactor = (Time.time / 31536000.0f) % 1.0f, // Year cycle
    humanImpact = 0.2f // Moderate human impact
};

JobHandle simulationHandle = simulationJob.Schedule(fishCount, 64);
JobHandle ecosystemHandle = ecosystemJob.Schedule(simulationHandle);
ecosystemHandle.Complete();

// Update ecosystem values
ecosystemHealth = health[0];
biodiversityIndex = diversity[0];
foodChainBalance = balance[0];

// Copy results to persistent arrays
newPositions.CopyTo(fishPositions);
newVelocities.CopyTo(fishVelocities);

// Cleanup
positions.Dispose();
velocities.Dispose();
sizes.Dispose();
species.Dispose();
newPositions.Dispose();
newVelocities.Dispose();
activeFish.Dispose();
tunaPop.Dispose();
```

```
    reefPop.Dispose();
    sharkPop.Dispose();
    turtlePop.Dispose();
    rayPop.Dispose();
    health.Dispose();
    diversity.Dispose();
    balance.Dispose();

    CreateMarineLifeObjects();
    StartEcosystemMonitoring();
}

int GetRandomMaldivianSpecies()
{
    // Weighted distribution based on Maldivian marine life
    float rand = UnityEngine.Random.Range(0f, 100f);
    if (rand < 40) return 0;      // Tuna (40%)
    else if (rand < 70) return 1; // Reef fish (30%)
    else if (rand < 85) return 2; // Small schooling fish (15%)
    else if (rand < 95) return 3; // Sharks (10%)
    else return 4;                // Turtles/Rays (5%)
}

void CreateMarineLifeObjects()
{
    // Create fish GameObjects
    for (int i = 0; i < maxFishCount; i++)
    {
        GameObject fish = GameObject.CreatePrimitive(PrimitiveType.Sphere);
        fish.name = $"Fish_{i}_{(MarineSpecies)fishSpecies[i]}";
        fish.transform.position = fishPositions[i];
```

```
            fish.transform.localScale = Vector3.one * fishSizes[i];

            // Add simple fish behavior
            fish.AddComponent<FishBehavior>().Initialize(fishSpecies[i], fishSizes[i]);

            fishObjects[i] = fish;
        }
    }

    void StartEcosystemMonitoring()
    {
        // Monitor ecosystem health
        InvokeRepeating("UpdateEcosystem", ecosystemUpdateInterval,
ecosystemUpdateInterval);
        InvokeRepeating("CheckConservationStatus", 30.0f, 30.0f);
    }

    void UpdateEcosystem()
    {
        if (!enableEcosystemSimulation) return;

        // Update ecosystem simulation
        Debug.Log($"Ecosystem Health: {ecosystemHealth:F2}, Biodiversity:
{biodiversityIndex:F2}, Balance: {foodChainBalance:F2}");
    }

    void CheckConservationStatus()
    {
        if (!preserveEndangeredSpecies) return;

        // Check for endangered species
```

```
    if (GetSpeciesPopulation(MarineSpecies.GreenSeaTurtle) < 10)
    {
        Debug.LogWarning("Green Sea Turtle population critically low!");
    }

    if (GetSpeciesPopulation(MarineSpecies.HawksbillTurtle) < 5)
    {
        Debug.LogWarning("Hawksbill Turtle population critically low!");
    }
}

int GetSpeciesPopulation(MarineSpecies species)
{
    // Count specific species
    int count = 0;
    for (int i = 0; i < fishSpecies.Length; i++)
    {
        if (fishSpecies[i] == (int)species) count++;
    }
    return count;
}

public void TriggerFeeding(Vector3 position, float range)
{
    // Attract fish to feeding area
    for (int i = 0; i < fishPositions.Length; i++)
    {
        float distance = Vector3.Distance(fishPositions[i], position);
        if (distance < range)
        {
            Vector3 direction = (position - fishPositions[i]).normalized;
```

```
                fishVelocities[i] += new float3(direction.x, direction.y, direction.z) * 2.0f;
            }
        }
    }

    public void ApplyFishingImpact(Vector3 position, float radius, int fishCaught)
    {
        if (!simulateTraditionalFishing) return;

        // Remove fish from area (sustainable fishing simulation)
        for (int i = 0; i < fishObjects.Length && fishCaught > 0; i++)
        {
            if (fishObjects[i] != null)
            {
                float distance = Vector3.Distance(fishObjects[i].transform.position,
position);
                if (distance < radius)
                {
                    Destroy(fishObjects[i]);
                    fishObjects[i] = null;
                    fishCaught--;
                }
            }
        }
    }

    public WildlifeSnapshot GetWildlifeSnapshot()
    {
        return new WildlifeSnapshot
        {
            total_marine_life = CountActiveFish(),
```

```
        ecosystem_health = ecosystemHealth,
        biodiversity_index = biodiversityIndex,
        food_chain_balance = foodChainBalance,
        endangered_species_count = CountEndangeredSpecies(),
        conservation_areas = CountConservationAreas()
    };
}

int CountActiveFish()
{
    int count = 0;
    for (int i = 0; i < fishObjects.Length; i++)
    {
        if (fishObjects[i] != null) count++;
    }
    return count;
}

int CountEndangeredSpecies()
{
    int count = 0;
    count += GetSpeciesPopulation(MarineSpecies.GreenSeaTurtle);
    count += GetSpeciesPopulation(MarineSpecies.HawksbillTurtle);
    return count;
}

int CountConservationAreas()
{
    // Count marine protected areas
    return respectMarineSanctuaries ? 5 : 0;
}
```

```
    void OnDestroy()
    {
        // Cleanup native arrays
        if (fishPositions.IsCreated) fishPositions.Dispose();
        if (fishVelocities.IsCreated) fishVelocities.Dispose();
        if (fishSizes.IsCreated) fishSizes.Dispose();
        if (fishSpecies.IsCreated) fishSpecies.Dispose();
    }

    [System.Serializable]
    public class WildlifeSnapshot
    {
        public int total_marine_life;
        public float ecosystem_health;
        public float biodiversity_index;
        public float food_chain_balance;
        public int endangered_species_count;
        public int conservation_areas;
    }
}

// Simple fish behavior component
public class FishBehavior : MonoBehaviour
{
    private int species;
    private float size;
    private Vector3 wanderDirection;
    private float nextWanderTime;

    public void Initialize(int fishSpecies, float fishSize)
```

```
    {
        species = fishSpecies;
        size = fishSize;
        wanderDirection = UnityEngine.Random.insideUnitSphere;
        nextWanderTime = Time.time + UnityEngine.Random.Range(2f, 8f);
    }

    void Update()
    {
        // Simple wandering behavior
        if (Time.time > nextWanderTime)
        {
            wanderDirection = UnityEngine.Random.insideUnitSphere;
            nextWanderTime = Time.time + UnityEngine.Random.Range(2f, 8f);
        }

        // Move fish
        transform.position += wanderDirection * size * 0.5f * Time.deltaTime;

        // Keep fish underwater
        if (transform.position.y > -1f)
        {
            transform.position = new Vector3(transform.position.x, -2f,
transform.position.z);
        }
    }
}
```

38. AchievementSystem.cs

```
using UnityEngine;
```

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class AchievementSystem : MonoBehaviour
{
    [BurstCompile]
    struct CulturalAchievementValidation : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> playerProgress;
        [ReadOnly] public NativeArray<bool> completionStatus;
        [ReadOnly] public NativeArray<int> culturalRequirements;
        [WriteOnly] public NativeArray<bool> newlyCompleted;
        [WriteOnly] public NativeArray<float> culturalSignificance;

        public void Execute(int index)
        {
            float progress = playerProgress[index];
            bool wasCompleted = completionStatus[index];
            int culturalReq = culturalRequirements[index];

            // Validate achievement completion
            bool isCompleted = progress >= 1.0f && culturalReq > 0;
            bool justCompleted = isCompleted && !wasCompleted;

            newlyCompleted[index] = justCompleted;
            culturalSignificance[index] = CalculateCulturalSignificance(culturalReq);
        }
```

```
    [BurstCompile]
    float CalculateCulturalSignificance(int requirement)
    {
        // Higher cultural requirements = higher significance
        return math.saturate(requirement / 10.0f);
    }
}

[BurstCompile]
struct MaldivianCulturalAchievements : IJob
{
    public NativeArray<bool> prayerAchievements;
    public NativeArray<bool> fishingAchievements;
    public NativeArray<bool> navigationAchievements;
    public NativeArray<bool> culturalKnowledgeAchievements;
    public NativeArray<bool> communityAchievements;
    public NativeArray<bool> environmentalAchievements;

    public NativeArray<float> overallCulturalProgress;
    public NativeArray<int> dhivehiLanguageUsage;
    public NativeArray<float> respectScores;

    public float totalPlayTime;
    public int islandsVisited;
    public int prayersAttended;
    public float culturalRespect;

    public void Execute()
    {
        ValidatePrayerAchievements();
        ValidateFishingAchievements();
```

```
            ValidateNavigationAchievements();
            ValidateCulturalKnowledgeAchievements();
            ValidateCommunityAchievements();
            ValidateEnvironmentalAchievements();
            CalculateOverallProgress();
        }

        [BurstCompile]
        void ValidatePrayerAchievements()
        {
            // Traditional Maldivian prayer achievements
            prayerAchievements[0] = prayersAttended >= 5;    // "Regular
Worshipper"
            prayerAchievements[1] = prayersAttended >= 25;   // "Devout
Practitioner"
            prayerAchievements[2] = prayersAttended >= 100;  // "Spiritual Guide"
            prayerAchievements[3] = culturalRespect > 0.8f;  // "Respectful Visitor"
        }

        [BurstCompile]
        void ValidateFishingAchievements()
        {
            // Traditional Maldivian fishing achievements
            // These would be tracked by the fishing system
            fishingAchievements[0] = true; // Placeholder for "Traditional Fisher"
            fishingAchievements[1] = true; // Placeholder for "Sustainable Harvester"
            fishingAchievements[2] = true; // Placeholder for "Master Navigator"
            fishingAchievements[3] = true; // Placeholder for "Ocean Guardian"
        }

        [BurstCompile]
```

```
    void ValidateNavigationAchievements()
    {
        // Traditional seafaring and navigation
        navigationAchievements[0] = islandsVisited >= 5;   // "Island Explorer"
        navigationAchievements[1] = islandsVisited >= 15;  // "Sea Navigator"
        navigationAchievements[2] = islandsVisited >= 30;  // "Ocean Master"
        navigationAchievements[3] = totalPlayTime > 3600;  // "Dedicated
Voyager"
    }

    [BurstCompile]
    void ValidateCulturalKnowledgeAchievements()
    {
        // Cultural learning and knowledge achievements
        int dhivehiUsage = dhivehiLanguageUsage[0];
        culturalKnowledgeAchievements[0] = dhivehiUsage > 10;   // "Language
Learner"
        culturalKnowledgeAchievements[1] = dhivehiUsage > 50;   // "Cultural
Student"
        culturalKnowledgeAchievements[2] = dhivehiUsage > 200;  // "Knowledge
Seeker"
        culturalKnowledgeAchievements[3] = culturalRespect > 0.9f; // "Cultural
Ambassador"
    }

    [BurstCompile]
    void ValidateCommunityAchievements()
    {
        // Community interaction achievements
        float avgRespect = respectScores[0];
        communityAchievements[0] = avgRespect > 0.6f;  // "Friendly Visitor"
```

```
            communityAchievements[1] = avgRespect > 0.75f; // "Community Helper"
            communityAchievements[2] = avgRespect > 0.85f; // "Respected Guest"
            communityAchievements[3] = avgRespect > 0.95f; // "Honored Friend"
        }

        [BurstCompile]
        void ValidateEnvironmentalAchievements()
        {
            // Environmental conservation achievements
            environmentalAchievements[0] = true; // Placeholder for "Ocean Cleaner"
            environmentalAchievements[1] = true; // Placeholder for "Coral Protector"
            environmentalAchievements[2] = true; // Placeholder for "Wildlife
Guardian"
            environmentalAchievements[3] = true; // Placeholder for "Eco Champion"
        }

        [BurstCompile]
        void CalculateOverallProgress()
        {
            // Calculate overall cultural progress
            int totalAchievements = 0;
            int completedAchievements = 0;

            for (int i = 0; i < prayerAchievements.Length; i++)
            {
                totalAchievements++;
                if (prayerAchievements[i]) completedAchievements++;
            }

            for (int i = 0; i < fishingAchievements.Length; i++)
            {
```

```
            totalAchievements++;
            if (fishingAchievements[i]) completedAchievements++;
        }

        for (int i = 0; i < navigationAchievements.Length; i++)
        {
            totalAchievements++;
            if (navigationAchievements[i]) completedAchievements++;
        }

        for (int i = 0; i < culturalKnowledgeAchievements.Length; i++)
        {
            totalAchievements++;
            if (culturalKnowledgeAchievements[i]) completedAchievements++;
        }

        for (int i = 0; i < communityAchievements.Length; i++)
        {
            totalAchievements++;
            if (communityAchievements[i]) completedAchievements++;
        }

        for (int i = 0; i < environmentalAchievements.Length; i++)
        {
            totalAchievements++;
            if (environmentalAchievements[i]) completedAchievements++;
        }

        overallCulturalProgress[0] = totalAchievements > 0 ?
(float)completedAchievements / totalAchievements : 0.0f;
    }
```

```
}

public static AchievementSystem Instance { get; private set; }

[Header("Achievement Settings")]
public bool enableAchievements = true;
public bool enableCulturalAchievements = true;
public bool enableProgressiveUnlocking = true;
public bool showAchievementNotifications = true;

[Header("Maldivian Cultural Focus")]
public bool emphasizePrayerAchievements = true;
public bool emphasizeFishingAchievements = true;
public bool emphasizeNavigationAchievements = true;
public bool emphasizeCommunityAchievements = true;
public bool emphasizeEnvironmentalAchievements = true;

[Header("Achievement Categories")]
public AchievementCategory[] achievementCategories;

private bool[] achievementCompleted;
private float[] achievementProgress;
private string[] achievementUnlockDates;
private int dhivehiUsageCount;
private float totalCulturalRespect;
private int totalPrayersAttended;
private float totalPlayTime;

[System.Serializable]
public class AchievementCategory
{
```

```
    public string categoryName;
    public string dhivehiName;
    public Sprite categoryIcon;
    public Color categoryColor;
    public bool isCultural;
    public int[] achievementIDs;

    public enum CategoryType
    {
        PrayerAndSpirituality,
        TraditionalFishing,
        NavigationAndSeafaring,
        CulturalKnowledge,
        CommunityInteraction,
        EnvironmentalConservation,
        Exploration,
        SocialHarmony,
        LanguageLearning,
        TraditionalCrafts
    }
}

[System.Serializable]
public class CulturalAchievement
{
    public int achievementID;
    public string achievementName;
    public string dhivehiName;
    public string description;
    public string culturalContext;
    public int culturalSignificance; // 1-10 scale
```

```
        public bool isSecret;
        public Sprite achievementIcon;
        public string[] unlockRequirements;
        public float progressMax;
        public bool unlocked;
        public string unlockDate;
    }

    void Awake()
    {
        Instance = this;
        InitializeAchievementSystem();
    }

    void InitializeAchievementSystem()
    {
        // Initialize achievement tracking
        int totalAchievements = 60; // 6 categories × 10 achievements each
        achievementCompleted = new bool[totalAchievements];
        achievementProgress = new float[totalAchievements];
        achievementUnlockDates = new string[totalAchievements];

        // Load achievement progress
        LoadAchievementProgress();

        // Initialize validation job
        int achievementCount = 24; // Sample subset
        var progress = new NativeArray<float>(achievementCount,
Allocator.TempJob);
        var completed = new NativeArray<bool>(achievementCount,
Allocator.TempJob);
```

```
        var culturalReqs = new NativeArray<int>(achievementCount,
Allocator.TempJob);
        var newlyCompleted = new NativeArray<bool>(achievementCount,
Allocator.TempJob);
        var culturalSignificance = new NativeArray<float>(achievementCount,
Allocator.TempJob);

        // Initialize with sample data
        for (int i = 0; i < achievementCount; i++)
        {
            progress[i] = UnityEngine.Random.Range(0.0f, 1.2f);
            completed[i] = progress[i] >= 1.0f;
            culturalReqs[i] = UnityEngine.Random.Range(1, 11);
        }

        var validationJob = new CulturalAchievementValidation
        {
            playerProgress = progress,
            completionStatus = completed,
            culturalRequirements = culturalReqs,
            newlyCompleted = newlyCompleted,
            culturalSignificance = culturalSignificance
        };

        // Maldivian cultural achievements
        var prayerAchievements = new NativeArray<bool>(4, Allocator.TempJob);
        var fishingAchievements = new NativeArray<bool>(4, Allocator.TempJob);
        var navigationAchievements = new NativeArray<bool>(4,
Allocator.TempJob);
        var culturalAchievements = new NativeArray<bool>(4, Allocator.TempJob);
```

```
        var communityAchievements = new NativeArray<bool>(4,
Allocator.TempJob);
        var environmentalAchievements = new NativeArray<bool>(4,
Allocator.TempJob);
        var overallProgress = new NativeArray<float>(1, Allocator.TempJob);
        var dhivehiUsage = new NativeArray<int>(1, Allocator.TempJob);
        var respectScores = new NativeArray<float>(1, Allocator.TempJob);

        // Initialize with player data
        dhivehiUsage[0] = dhivehiUsageCount;
        respectScores[0] = totalCulturalRespect;
        totalPlayTime = Time.time;

        var culturalJob = new MaldivianCulturalAchievements
        {
            prayerAchievements = prayerAchievements,
            fishingAchievements = fishingAchievements,
            navigationAchievements = navigationAchievements,
            culturalKnowledgeAchievements = culturalAchievements,
            communityAchievements = communityAchievements,
            environmentalAchievements = environmentalAchievements,
            overallCulturalProgress = overallProgress,
            dhivehiLanguageUsage = dhivehiUsage,
            respectScores = respectScores,
            totalPlayTime = totalPlayTime,
            islandsVisited = 15, // Placeholder
            prayersAttended = totalPrayersAttended,
            culturalRespect = totalCulturalRespect
        };
```

```
        JobHandle validationHandle = validationJob.Schedule(achievementCount,
8);
        JobHandle culturalHandle = culturalJob.Schedule(validationHandle);
        culturalHandle.Complete();

        // Process newly completed achievements
        for (int i = 0; i < newlyCompleted.Length; i++)
        {
            if (newlyCompleted[i])
            {
                OnAchievementCompleted(i, culturalSignificance[i]);
            }
        }

        // Update cultural progress
        float culturalProgress = overallProgress[0];
        Debug.Log($"Overall Cultural Progress: {culturalProgress:P}");

        // Cleanup
        progress.Dispose();
        completed.Dispose();
        culturalReqs.Dispose();
        newlyCompleted.Dispose();
        culturalSignificance.Dispose();
        prayerAchievements.Dispose();
        fishingAchievements.Dispose();
        navigationAchievements.Dispose();
        culturalAchievements.Dispose();
        communityAchievements.Dispose();
        environmentalAchievements.Dispose();
        overallProgress.Dispose();
```

```
        dhivehiUsage.Dispose();
        respectScores.Dispose();

        StartAchievementTracking();
    }

    void LoadAchievementProgress()
    {
        // Load from PlayerPrefs in production
        for (int i = 0; i < achievementCompleted.Length; i++)
        {
            achievementCompleted[i] = false;
            achievementProgress[i] = 0.0f;
        }
    }

    void StartAchievementTracking()
    {
        // Track achievement progress
        InvokeRepeating("UpdateAchievementProgress", 10.0f, 30.0f);
        InvokeRepeating("CheckMilestoneAchievements", 60.0f, 300.0f);
    }

    public void UpdateAchievementProgress(int achievementID, float progress)
    {
        if (achievementID < 0 || achievementID >= achievementProgress.Length)
return;

        achievementProgress[achievementID] = Mathf.Clamp01(progress);
```

```
        if (achievementProgress[achievementID] >= 1.0f &&
!achievementCompleted[achievementID])
        {
            CompleteAchievement(achievementID);
        }
    }

    public void CompleteAchievement(int achievementID)
    {
        if (achievementCompleted[achievementID]) return;

        achievementCompleted[achievementID] = true;
        achievementUnlockDates[achievementID] =
System.DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");

        OnAchievementCompleted(achievementID,
GetCulturalSsignificance(achievementID));
    }

    void OnAchievementCompleted(int achievementID, float culturalSignificance)
    {
        Debug.Log($"Achievement Unlocked:
{GetAchievementName(achievementID)} (Cultural Significance:
{culturalSignificance:F2})");

        if (showAchievementNotifications)
        {
            ShowAchievementNotification(achievementID, culturalSignificance);
        }

        // Trigger rewards
```

```
        GrantAchievementRewards(achievementID, culturalSignificance);

        // Check for milestone achievements
        CheckMilestoneAchievements();
    }

    void ShowAchievementNotification(int achievementID, float
culturalSignificance)
    {
        string achievementName = GetAchievementName(achievementID);
        string message = GetDhivehiAchievementMessage(achievementID,
culturalSignificance);

        // Show through UI system
        if (UISystem.Instance != null)
        {
            UISystem.Instance.ShowCulturalNotification(message, 5.0f);
        }
    }

    string GetDhivehiAchievementMessage(int achievementID, float
culturalSignificance)
    {
        string baseMessage = $"Ô¨Å√úÔ¨Å¬∂Ô¨Å√ßÔ¨Å√òÔ¨Å√áÔ¨Å,àû!
{GetAchievementName(achievementID)}"; // "Achievement unlocked!"

        if (culturalSignificance > 0.7f)
        {
            baseMessage += " - Ô¨Å√†Ô¨Å¬ÆÔ¨Å√ßÔ¨Å,Ñ¢
Ô¨Å√†Ô¨Å¬∂Ô¨Å√ÖÔ¨Å¬©Ô¨Å√úÔ¨Å,Ñ¢
Ô¨Å√†Ô¨Å¬∂Ô¨Å√ÖÔ¨Å¬©Ô¨Å√úÔ¨Å,Ñ¢"; // "Great cultural achievement!"
```

```
        }

        return baseMessage;
    }

    void GrantAchievementRewards(int achievementID, float culturalSignificance)
    {
        // Grant experience, currency, or unlocks based on cultural significance
        int experienceReward = Mathf.RoundToInt(culturalSignificance * 1000);
        int culturalPoints = Mathf.RoundToInt(culturalSignificance * 100);

        if (SkillSystem.Instance != null)
        {
            SkillSystem.Instance.GainSkillExperience("Cultural Knowledge",
experienceReward, true);
        }

        Debug.Log($"Granted {experienceReward} experience and {culturalPoints}
cultural points");
    }

    void CheckMilestoneAchievements()
    {
        // Check for meta-achievements
        int completedCount = 0;
        for (int i = 0; i < achievementCompleted.Length; i++)
        {
            if (achievementCompleted[i]) completedCount++;
        }

        if (completedCount >= 10)
```

```
        {
            Debug.Log("Milestone: Cultural Explorer - 10 achievements completed!");
        }

        if (completedCount >= 25)
        {
            Debug.Log("Milestone: Cultural Ambassador - 25 achievements
completed!");
        }

        if (completedCount >= 50)
        {
            Debug.Log("Milestone: Cultural Master - 50 achievements completed!");
        }
    }

    public void TrackDhivehiUsage(int usageCount)
    {
        dhivehiUsageCount += usageCount;
    }

    public void TrackCulturalRespect(float respectScore)
    {
        totalCulturalRespect = Mathf.Max(totalCulturalRespect, respectScore);
    }

    public void TrackPrayerAttendance()
    {
        totalPrayersAttended++;
    }
```

```
    string GetAchievementName(int achievementID)
    {
        // Return achievement name based on ID
        return $"Achievement_{achievementID}";
    }

    float GetCulturalSignificance(int achievementID)
    {
        // Return cultural significance score
        return (achievementID % 10) / 10.0f;
    }

    public AchievementSnapshot GetAchievementSnapshot()
    {
        int completedCount = 0;
        for (int i = 0; i < achievementCompleted.Length; i++)
        {
            if (achievementCompleted[i]) completedCount++;
        }

        return new AchievementSnapshot
        {
            total_achievements = achievementCompleted.Length,
            completed_achievements = completedCount,
            completion_percentage = (float)completedCount /
achievementCompleted.Length,
            cultural_achievements = CountCulturalAchievements(),
            dhivehi_usage_count = dhivehiUsageCount,
            total_prayers_attended = totalPrayersAttended,
            average_cultural_respect = totalCulturalRespect
        };
```

```
}

int CountCulturalAchievements()
{
    int count = 0;
    for (int i = 0; i < achievementCompleted.Length; i++)
    {
        if (achievementCompleted[i] && IsCulturalAchievement(i))
        {
            count++;
        }
    }
    return count;
}

bool IsCulturalAchievement(int achievementID)
{
    return achievementID < 30; // First 30 are cultural
}

[System.Serializable]
public class AchievementSnapshot
{
    public int total_achievements;
    public int completed_achievements;
    public float completion_percentage;
    public int cultural_achievements;
    public int dhivehi_usage_count;
    public int total_prayers_attended;
    public float average_cultural_respect;
}
```

```
}
```

39. TutorialSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class TutorialSystem : MonoBehaviour
{
    [BurstCompile]
    struct ContextualTutorialSystem : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> playerPositionHistory;
        [ReadOnly] public NativeArray<float> timeInAreas;
        [ReadOnly] public NativeArray<int> interactionCounts;
        [ReadOnly] public NativeArray<bool> systemUsages;
        [WriteOnly] public NativeArray<bool> tutorialTriggers;
        [WriteOnly] public NativeArray<int> tutorialPriorities;

        public float currentTime;
        public float3 playerPosition;
        public int currentIsland;
        public bool isFirstTime;
```

```
    public void Execute(int index)
    {
        float3 position = playerPositionHistory[index];
        float timeInArea = timeInAreas[index];
        int interactions = interactionCounts[index];
        bool systemUsed = systemUsages[index];

        // Determine if tutorial should trigger
        bool shouldTrigger = ShouldTriggerTutorial(index, position, timeInArea,
interactions, systemUsed);
        int priority = CalculateTutorialPriority(index, shouldTrigger);

        tutorialTriggers[index] = shouldTrigger;
        tutorialPriorities[index] = priority;
    }

    [BurstCompile]
    bool ShouldTriggerTutorial(int tutorialID, float3 position, float timeInArea, int
interactions, bool systemUsed)
    {
        // Context-aware tutorial triggering
        return tutorialID switch
        {
            0 => isFirstTime,                                // Basic movement
            1 => currentIsland >= 0 && timeInArea > 30.0f,     // Island exploration
            2 => interactions == 0 && currentTime > 60.0f,     // First interaction
prompt
            3 => !systemUsed && currentTime > 120.0f,          // System usage
prompt
```

```
            4 => currentIsland == 0 && timeInArea > 60.0f,     // Cultural
introduction
            5 => true,                                  // Prayer time awareness
            6 => interactions > 5,                      // Advanced interactions
            7 => currentIsland > 5,                     // Multi-island navigation
            8 => timeInArea > 300.0f,                   // Long-term guidance
            _ => false
        };
    }

    [BurstCompile]
    int CalculateTutorialPriority(int tutorialID, bool shouldTrigger)
    {
        if (!shouldTrigger) return 0;

        return tutorialID switch
        {
            0 => 10, // Critical - first time user
            1 => 8,  // High - island exploration
            2 => 7,  // High - interaction guidance
            3 => 6,  // Medium - system usage
            4 => 9,  // Very High - cultural respect
            5 => 8,  // High - prayer awareness
            6 => 5,  // Medium - advanced features
            7 => 4,  // Low - experienced user
            8 => 3,  // Low - optional guidance
            _ => 1
        };
    }
}
```

```
[BurstCompile]
struct MaldivianCulturalTutorial : IJob
{
    public NativeArray<bool> culturalTutorialCompleted;
    public NativeArray<float> culturalUnderstanding;
    public NativeArray<int> respectfulBehaviors;
    public NativeArray<int> prayerParticipation;
    public NativeArray<int> traditionalActivities;

    public NativeArray<float> overallCulturalLearning;
    public NativeArray<bool> isRespectfulVisitor;

    public float totalPlayTime;
    public int dhivehiPhrasesLearned;
    public int culturalMistakes;
    public int culturalSuccesses;

    public void Execute()
    {
        EvaluateCulturalLearning();
        CalculateRespectfulVisitorStatus();
        GenerateCulturalRecommendations();
    }

    [BurstCompile]
    void EvaluateCulturalLearning()
    {
        // Calculate cultural understanding based on behaviors
        float prayerScore = prayerParticipation[0] * 0.25f;
        float respectScore = respectfulBehaviors[0] * 0.30f;
        float activityScore = traditionalActivities[0] * 0.20f;
```

```
        float languageScore = math.saturate(dhivehiPhrasesLearned / 20.0f) *
0.15f;
        float timeScore = math.saturate(totalPlayTime / 1800.0f) * 0.10f; // 30
minutes

        float understanding = prayerScore + respectScore + activityScore +
languageScore + timeScore;
        culturalUnderstanding[0] = math.saturate(understanding);

        // Calculate learning rate
        float learningRate = culturalSuccesses / math.max(1, culturalSuccesses +
culturalMistakes);
        overallCulturalLearning[0] = understanding * learningRate;
    }

    [BurstCompile]
    void CalculateRespectfulVisitorStatus()
    {
        // Determine if player is a respectful visitor
        bool highRespect = respectfulBehaviors[0] >= 10;
        bool activeParticipation = prayerParticipation[0] >= 5;
        bool culturalEngagement = traditionalActivities[0] >= 3;
        bool languageEffort = dhivehiPhrasesLearned >= 5;
        bool lowMistakes = culturalMistakes < culturalSuccesses;

        isRespectfulVisitor[0] = highRespect && activeParticipation &&
                                 culturalEngagement && languageEffort && lowMistakes;
    }

    [BurstCompile]
    void GenerateCulturalRecommendations()
```

```
    {
        // Generate personalized cultural learning recommendations
        if (prayerParticipation[0] < 3)
        {
            // Recommend prayer participation
        }

        if (respectfulBehaviors[0] < 5)
        {
            // Recommend respectful behavior learning
        }

        if (dhivehiPhrasesLearned < 10)
        {
            // Recommend language learning
        }
    }
}

public static TutorialSystem Instance { get; private set; }

[Header("Tutorial Settings")]
public bool enableTutorials = true;
public bool enableCulturalTutorials = true;
public bool enableContextualTutorials = true;
public bool enableAdaptiveDifficulty = true;
public float tutorialDelay = 2.0f;

[Header("Cultural Learning")]
public bool emphasizeCulturalRespect = true;
public bool teachTraditionalPractices = true;
```

```
public bool explainReligiousSignificance = true;
public bool promoteEnvironmentalAwareness = true;

[Header("Tutorial Categories")]
public TutorialCategory[] tutorialCategories;

private bool[] tutorialCompleted;
private bool[] tutorialShown;
private float[] tutorialProgress;
private int currentTutorialID;
private bool isFirstTimeUser;
private float startTime;

[System.Serializable]
public class TutorialCategory
{
    public string categoryName;
    public string dhivehiName;
    public Color categoryColor;
    public bool isCultural;
    public Tutorial[] tutorials;

    public enum CategoryType
    {
        BasicControls,
        IslandExploration,
        CulturalAwareness,
        ReligiousPractices,
        TraditionalActivities,
        EnvironmentalConservation,
        CommunityInteraction,
```

```
        AdvancedFeatures
    }
}

[System.Serializable]
public class Tutorial
{
    public int tutorialID;
    public string tutorialName;
    public string description;
    public string culturalContext;
    public bool isContextual;
    public bool canSkip;
    public float displayDuration;
    public string[] steps;
    public Sprite[] tutorialImages;
    public AudioClip[] tutorialAudio;
}

void Awake()
{
    Instance = this;
    InitializeTutorialSystem();
}

void InitializeTutorialSystem()
{
    // Initialize tutorial tracking
    int totalTutorials = 40;
    tutorialCompleted = new bool[totalTutorials];
    tutorialShown = new bool[totalTutorials];
```

```
        tutorialProgress = new float[totalTutorials];

        // Check if first time user
        isFirstTimeUser = PlayerPrefs.GetInt("FirstTimeUser", 1) == 1;
        startTime = Time.time;

        // Initialize contextual tutorial system
        int historySize = 10;
        var positionHistory = new NativeArray<float3>(historySize,
Allocator.TempJob);
        var timeInAreas = new NativeArray<float>(historySize, Allocator.TempJob);
        var interactions = new NativeArray<int>(historySize, Allocator.TempJob);
        var systemUsages = new NativeArray<bool>(historySize,
Allocator.TempJob);
        var triggers = new NativeArray<bool>(historySize, Allocator.TempJob);
        var priorities = new NativeArray<int>(historySize, Allocator.TempJob);

        // Initialize with sample data
        for (int i = 0; i < historySize; i++)
        {
            positionHistory[i] = UnityEngine.Random.insideUnitSphere * 20.0f;
            timeInAreas[i] = UnityEngine.Random.Range(0.0f, 300.0f);
            interactions[i] = UnityEngine.Random.Range(0, 10);
            systemUsages[i] = UnityEngine.Random.Range(0, 2) == 1;
        }

        var contextualJob = new ContextualTutorialSystem
        {
            playerPositionHistory = positionHistory,
            timeInAreas = timeInAreas,
            interactionCounts = interactions,
```

```
            systemUsages = systemUsages,
            tutorialTriggers = triggers,
            tutorialPriorities = priorities,
            currentTime = Time.time - startTime,
            playerPosition = Vector3.zero,
            currentIsland = -1,
            isFirstTime = isFirstTimeUser
        };

        // Initialize cultural tutorial system
        var culturalCompleted = new NativeArray<bool>(6, Allocator.TempJob);
        var culturalUnderstanding = new NativeArray<float>(1, Allocator.TempJob);
        var respectfulBehaviors = new NativeArray<int>(1, Allocator.TempJob);
        var prayerParticipation = new NativeArray<int>(1, Allocator.TempJob);
        var traditionalActivities = new NativeArray<int>(1, Allocator.TempJob);
        var overallLearning = new NativeArray<float>(1, Allocator.TempJob);
        var respectfulVisitor = new NativeArray<bool>(1, Allocator.TempJob);

        // Initialize with player data
        culturalCompleted[0] = PlayerPrefs.GetInt("CulturalTutorial_Basic", 0) == 1;
        culturalCompleted[1] = PlayerPrefs.GetInt("CulturalTutorial_Prayer", 0) ==
1;
        culturalCompleted[2] = PlayerPrefs.GetInt("CulturalTutorial_Fishing", 0) ==
1;
        culturalCompleted[3] = PlayerPrefs.GetInt("CulturalTutorial_Navigation", 0)
== 1;
        culturalCompleted[4] = PlayerPrefs.GetInt("CulturalTutorial_Community", 0)
== 1;
        culturalCompleted[5] = PlayerPrefs.GetInt("CulturalTutorial_Environment",
0) == 1;
```

```
        var culturalJob = new MaldivianCulturalTutorial
        {
            culturalTutorialCompleted = culturalCompleted,
            culturalUnderstanding = culturalUnderstanding,
            respectfulBehaviors = respectfulBehaviors,
            prayerParticipation = prayerParticipation,
            traditionalActivities = traditionalActivities,
            overallCulturalLearning = overallLearning,
            isRespectfulVisitor = respectfulVisitor,
            totalPlayTime = Time.time - startTime,
            dhivehiPhrasesLearned = 0,
            culturalMistakes = 0,
            culturalSuccesses = 0
        };

        JobHandle contextualHandle = contextualJob.Schedule(historySize, 4);
        JobHandle culturalHandle = culturalJob.Schedule(contextualHandle);
        culturalHandle.Complete();

        // Process tutorial triggers
        ProcessTutorialTriggers(triggers, priorities);

        // Process cultural learning
        float understanding = culturalUnderstanding[0];
        bool isRespectful = respectfulVisitor[0];

        Debug.Log($"Cultural Understanding: {understanding:P}, Respectful Visitor:
{isRespectful}");

        // Cleanup
        positionHistory.Dispose();
```

```
        timeInAreas.Dispose();
        interactions.Dispose();
        systemUsages.Dispose();
        triggers.Dispose();
        priorities.Dispose();
        culturalCompleted.Dispose();
        culturalUnderstanding.Dispose();
        respectfulBehaviors.Dispose();
        prayerParticipation.Dispose();
        traditionalActivities.Dispose();
        overallLearning.Dispose();
        respectfulVisitor.Dispose();

        StartTutorialMonitoring();
    }

    void ProcessTutorialTriggers(NativeArray<bool> triggers, NativeArray<int>
priorities)
    {
        // Find highest priority tutorial to trigger
        int highestPriority = 0;
        int tutorialToTrigger = -1;

        for (int i = 0; i < triggers.Length; i++)
        {
            if (triggers[i] && priorities[i] > highestPriority)
            {
                highestPriority = priorities[i];
                tutorialToTrigger = i;
            }
        }
```

```
    if (tutorialToTrigger >= 0 && !tutorialShown[tutorialToTrigger])
    {
        TriggerTutorial(tutorialToTrigger, highestPriority);
    }
}

void StartTutorialMonitoring()
{
    // Monitor for tutorial triggers
    InvokeRepeating("CheckTutorialTriggers", tutorialDelay, 5.0f);
    InvokeRepeating("UpdateCulturalLearning", 30.0f, 60.0f);
}

void CheckTutorialTriggers()
{
    if (!enableContextualTutorials) return;

    // Check various conditions for tutorial triggers
    float timePlayed = Time.time - startTime;

    if (isFirstTimeUser && timePlayed > 10.0f && !tutorialShown[0])
    {
        TriggerTutorial(0, 10); // High priority first tutorial
    }

    if (timePlayed > 300.0f && !tutorialShown[4])
    {
        TriggerTutorial(4, 9); // Cultural introduction
    }
}
```

```
    void TriggerTutorial(int tutorialID, int priority)
    {
        if (!enableTutorials || tutorialCompleted[tutorialID] || tutorialShown[tutorialID])
return;

        currentTutorialID = tutorialID;
        tutorialShown[tutorialID] = true;

        StartTutorial(tutorialID);
    }

    void StartTutorial(int tutorialID)
    {
        // Get tutorial data
        Tutorial tutorial = GetTutorialByID(tutorialID);
        if (tutorial == null) return;

        Debug.Log($"Starting Tutorial: {tutorial.tutorialName}");

        // Show tutorial UI
        ShowTutorialUI(tutorial);

        // Play tutorial audio if available
        if (tutorial.tutorialAudio != null && tutorial.tutorialAudio.Length > 0)
        {
            PlayTutorialAudio(tutorial.tutorialAudio[0]);
        }

        // Start tutorial progression
        StartCoroutine(TutorialProgression(tutorial));
```

```
    }

    Tutorial GetTutorialByID(int tutorialID)
    {
        // Find tutorial by ID
        foreach (var category in tutorialCategories)
        {
            foreach (var tutorial in category.tutorials)
            {
                if (tutorial.tutorialID == tutorialID)
                    return tutorial;
            }
        }
        return null;
    }

    void ShowTutorialUI(Tutorial tutorial)
    {
        // Display tutorial through UI system
        if (UISystem.Instance != null)
        {
            string message = FormatTutorialMessage(tutorial);
            UISystem.Instance.ShowTutorialPanel(message,
tutorial.displayDuration);
        }
    }

    string FormatTutorialMessage(Tutorial tutorial)
    {
        string message = $"<b>{tutorial.tutorialName}</b>\n\n{tutorial.description}";
```

```
    if (enableCulturalTutorials && !string.IsNullOrEmpty(tutorial.culturalContext))
    {
        message += $"\n\n<i>Cultural Context: {tutorial.culturalContext}</i>";
    }

    if (tutorial.steps != null && tutorial.steps.Length > 0)
    {
        message += "\n\n<b>Steps:</b>";
        for (int i = 0; i < tutorial.steps.Length; i++)
        {
            message += $"\n{i + 1}. {tutorial.steps[i]}";
        }
    }

    return message;
}

void PlayTutorialAudio(AudioClip audioClip)
{
    // Play tutorial audio with cultural sensitivity
    if (AudioSystem.Instance != null)
    {
        AudioSystem.Instance.PlayTutorialAudio(audioClip);
    }
}

System.Collections.IEnumerator TutorialProgression(Tutorial tutorial)
{
    float elapsedTime = 0.0f;
    int currentStep = 0;
```

```
    while (elapsedTime < tutorial.displayDuration)
    {
        yield return null;
        elapsedTime += Time.deltaTime;

        // Check for step completion
        if (tutorial.steps != null && currentStep < tutorial.steps.Length)
        {
            if (CheckStepCompletion(tutorial.tutorialID, currentStep))
            {
                currentStep++;
                OnTutorialStepCompleted(tutorial.tutorialID, currentStep);
            }
        }
    }

    // Complete tutorial
    CompleteTutorial(tutorial.tutorialID);
}

bool CheckStepCompletion(int tutorialID, int stepIndex)
{
    // Check if current tutorial step is completed
    // Implementation depends on tutorial type
    return false;
}

void OnTutorialStepCompleted(int tutorialID, int stepIndex)
{
    Debug.Log($"Tutorial {tutorialID} step {stepIndex} completed");
    tutorialProgress[tutorialID] = (float)stepIndex / GetTutorialSteps(tutorialID);
```

```
}

int GetTutorialSteps(int tutorialID)
{
    Tutorial tutorial = GetTutorialByID(tutorialID);
    return tutorial?.steps?.Length ?? 1;
}

void CompleteTutorial(int tutorialID)
{
    tutorialCompleted[tutorialID] = true;
    tutorialProgress[tutorialID] = 1.0f;

    Debug.Log($"Tutorial {tutorialID} completed!");

    // Grant rewards
    GrantTutorialRewards(tutorialID);

    // Check for cultural tutorial completion
    if (IsCulturalTutorial(tutorialID))
    {
        OnCulturalTutorialCompleted(tutorialID);
    }
}

void GrantTutorialRewards(int tutorialID)
{
    // Grant experience or unlocks
    int experienceReward = 100;
    if (IsCulturalTutorial(tutorialID))
    {
```

```
            experienceReward *= 2; // Double reward for cultural tutorials
        }

        if (SkillSystem.Instance != null)
        {
            SkillSystem.Instance.GainSkillExperience("General Knowledge",
experienceReward, false);
        }
    }

    bool IsCulturalTutorial(int tutorialID)
    {
        // Check if tutorial is cultural
        foreach (var category in tutorialCategories)
        {
            if (category.isCultural)
            {
                foreach (var tutorial in category.tutorials)
                {
                    if (tutorial.tutorialID == tutorialID)
                        return true;
                }
            }
        }
        return false;
    }

    void OnCulturalTutorialCompleted(int tutorialID)
    {
        Debug.Log($"Cultural tutorial completed: {tutorialID}");
```

```
        // Update cultural understanding
        if (UISystem.Instance != null)
        {
            string message = "Ô¨Å√úÔ¨Å¬∂Ô¨Å√ßÔ¨Å√òÔ¨Å√áÔ¨Å,àû
Ô¨Å√†Ô¨Å¬ÆÔ¨Å√ßÔ¨Å,Ñ¢! Ô¨Å√ÜÔ¨Å¬∂Ô¨Å√ÖÔ¨Å¬©Ô¨Å√úÔ¨Å,Ñ¢
Ô¨Å√†Ô¨Å¬∂Ô¨Å√ÖÔ¨Å¬©Ô¨Å√úÔ¨Å,Ñ¢
Ô¨Å√ÜÔ¨Å¬∂Ô¨Å√ÖÔ¨Å¬©Ô¨Å√úÔ¨Å,Ñ¢ Ô¨Å√†Ô¨Å¬ÆÔ¨Å√ßÔ¨Å,Ñ¢"; //
"Achievement! Cultural understanding increased!"
            UISystem.Instance.ShowCulturalNotification(message, 3.0f);
        }
    }

    void UpdateCulturalLearning()
    {
        if (!enableCulturalTutorials) return;

        // Update cultural learning progress
        Debug.Log("Cultural learning progress updated");
    }

    public void SkipTutorial(int tutorialID)
    {
        if (!tutorialShown[tutorialID]) return;

        Tutorial tutorial = GetTutorialByID(tutorialID);
        if (tutorial != null && tutorial.canSkip)
        {
            CompleteTutorial(tutorialID);
        }
    }
```

```
public TutorialSnapshot GetTutorialSnapshot()
{
    int completedCount = 0;
    int culturalCompleted = 0;

    for (int i = 0; i < tutorialCompleted.Length; i++)
    {
        if (tutorialCompleted[i])
        {
            completedCount++;
            if (IsCulturalTutorial(i)) culturalCompleted++;
        }
    }

    return new TutorialSnapshot
    {
        total_tutorials = tutorialCompleted.Length,
        completed_tutorials = completedCount,
        cultural_tutorials_completed = culturalCompleted,
        first_time_user = isFirstTimeUser,
        adaptive_learning = enableAdaptiveDifficulty,
        cultural_emphasis = enableCulturalTutorials
    };
}

[System.Serializable]
public class TutorialSnapshot
{
    public int total_tutorials;
    public int completed_tutorials;
    public int cultural_tutorials_completed;
```

```
        public bool first_time_user;
        public bool adaptive_learning;
        public bool cultural_emphasis;
    }
}
```

40. LocalizationSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Text;

[BurstCompile]
public class LocalizationSystem : MonoBehaviour
{
    [BurstCompile]
    struct DhivehiTextProcessing : IJobParallelFor
    {
        [ReadOnly] public NativeArray<int> textIDs;
        [ReadOnly] public NativeArray<bool> isCulturalText;
        [ReadOnly] public NativeArray<int> culturalContextLevels;
        [WriteOnly] public NativeArray<float3> textLayoutData;
        [WriteOnly] public NativeArray<bool> requiresRTL;
        [WriteOnly] public NativeArray<int> fontSizeAdjustments;

        public void Execute(int index)
```

```
{
    int textID = textIDs[index];
    bool cultural = isCulturalText[index];
    int contextLevel = culturalContextLevels[index];

    // Process Dhivehi text layout
    float3 layout = CalculateTextLayout(textID, cultural);
    bool rtl = ShouldUseRTL(textID);
    int fontSize = CalculateFontSize(contextLevel);

    textLayoutData[index] = layout;
    requiresRTL[index] = rtl;
    fontSizeAdjustments[index] = fontSize;
}

[BurstCompile]
float3 CalculateTextLayout(int textID, bool isCultural)
{
    // Calculate optimal text layout for Dhivehi
    float width = isCultural ? 0.9f : 0.8f; // More space for cultural text
    float height = 0.1f;
    float lineSpacing = isCultural ? 1.2f : 1.0f;

    return new float3(width, height, lineSpacing);
}

[BurstCompile]
bool ShouldUseRTL(int textID)
{
    // Dhivehi is written right-to-left
    return textID >= 1000 && textID < 2000; // Dhivehi text ID range
```

```
    }

    [BurstCompile]
    int CalculateFontSize(int contextLevel)
    {
        // Adjust font size based on cultural importance
        return contextLevel switch
        {
            0 => 14, // Normal text
            1 => 16, // Important cultural context
            2 => 18, // Religious text
            3 => 20, // Traditional practices
            _ => 14
        };
    }
}

[BurstCompile]
struct MaldivianCulturalLocalization : IJob
{
    public NativeArray<int> prayerTimeNames;
    public NativeArray<int> islandNames;
    public NativeArray<int> fishingTerms;
    public NativeArray<int> traditionalTitles;
    public NativeArray<int> respectfulPhrases;

    public NativeArray<bool> culturalAuthenticity;
    public NativeArray<float> pronunciationAccuracy;
    public NativeArray<int> contextualAppropriateness;

    public int currentPrayerTime;
```

```
        public int currentIsland;
        public bool isFormalContext;
        public bool isReligiousContext;

        public void Execute()
        {
            ValidatePrayerTimeTranslations();
            ValidateIslandNames();
            ValidateFishingTerminology();
            ValidateTraditionalTitles();
            ValidateRespectfulLanguage();
            CalculateCulturalAuthenticity();
        }

        [BurstCompile]
        void ValidatePrayerTimeTranslations()
        {
            // Validate Dhivehi prayer time names
            for (int i = 0; i < prayerTimeNames.Length; i++)
            {
                int nameID = prayerTimeNames[i];
                bool isAuthentic = IsAuthenticPrayerName(nameID, i);
                culturalAuthenticity[i] = isAuthentic;

                if (isAuthentic)
                {
                    pronunciationAccuracy[i] = 0.95f; // High accuracy for authentic
terms
                }
            }
        }
```

```
        [BurstCompile]
        void ValidateIslandNames()
        {
            // Validate traditional island names
            for (int i = 0; i < islandNames.Length; i++)
            {
                int nameID = islandNames[i];
                bool isTraditional = IsTraditionalIslandName(nameID, i);
                culturalAuthenticity[prayerTimeNames.Length + i] = isTraditional;

                if (isTraditional)
                {
                    pronunciationAccuracy[prayerTimeNames.Length + i] = 0.90f;
                }
            }
        }

        [BurstCompile]
        void ValidateFishingTerminology()
        {
            // Validate traditional fishing terms
            for (int i = 0; i < fishingTerms.Length; i++)
            {
                int termID = fishingTerms[i];
                bool isTraditional = IsTraditionalFishingTerm(termID, i);
                culturalAuthenticity[prayerTimeNames.Length + islandNames.Length +
i] = isTraditional;

                if (isTraditional)
                {
```

```
                pronunciationAccuracy[prayerTimeNames.Length +
islandNames.Length + i] = 0.85f;
            }
        }
    }

    [BurstCompile]
    void ValidateTraditionalTitles()
    {
        // Validate respectful titles and honorifics
        for (int i = 0; i < traditionalTitles.Length; i++)
        {
            int titleID = traditionalTitles[i];
            bool isRespectful = IsRespectfulTitle(titleID, i);
            culturalAuthenticity[prayerTimeNames.Length + islandNames.Length +
fishingTerms.Length + i] = isRespectful;

            if (isRespectful)
            {
                pronunciationAccuracy[prayerTimeNames.Length +
islandNames.Length + fishingTerms.Length + i] = 0.95f;
            }
        }
    }

    [BurstCompile]
    void ValidateRespectfulLanguage()
    {
        // Validate respectful language usage
        for (int i = 0; i < respectfulPhrases.Length; i++)
        {
```

```
            int phraseID = respectfulPhrases[i];
            bool isAppropriate = IsContextuallyAppropriate(phraseID,
isFormalContext, isReligiousContext);
            culturalAuthenticity[prayerTimeNames.Length + islandNames.Length +
fishingTerms.Length + traditionalTitles.Length + i] = isAppropriate;

            if (isAppropriate)
            {
                pronunciationAccuracy[prayerTimeNames.Length +
islandNames.Length + fishingTerms.Length + traditionalTitles.Length + i] = 0.90f;
            }
        }
    }

    [BurstCompile]
    void CalculateCulturalAuthenticity()
    {
        // Calculate overall cultural authenticity score
        int totalItems = culturalAuthenticity.Length;
        int authenticItems = 0;

        for (int i = 0; i < totalItems; i++)
        {
            if (culturalAuthenticity[i]) authenticItems++;
        }

        float authenticityScore = (float)authenticItems / totalItems;

        // Apply contextual appropriateness
        for (int i = 0; i < contextualAppropriateness.Length; i++)
        {
```

```
            contextualAppropriateness[i] = Mathf.RoundToInt(authenticityScore *
10);
        }
    }

    [BurstCompile]
    bool IsAuthenticPrayerName(int nameID, int prayerIndex)
    {
        // Validate authentic Dhivehi prayer names
        return nameID >= 1000 && nameID < 1100; // Authentic prayer name ID
range
    }

    [BurstCompile]
    bool IsTraditionalIslandName(int nameID, int islandIndex)
    {
        // Validate traditional island names
        return nameID >= 2000 && nameID < 2100; // Traditional name ID range
    }

    [BurstCompile]
    bool IsTraditionalFishingTerm(int termID, int termIndex)
    {
        // Validate traditional fishing terminology
        return termID >= 3000 && termID < 3100; // Traditional fishing term ID
range
    }

    [BurstCompile]
    bool IsRespectfulTitle(int titleID, int titleIndex)
    {
```

```
            // Validate respectful titles and honorifics
            return titleID >= 4000 && titleID < 4100; // Respectful title ID range
        }

        [BurstCompile]
        bool IsContextuallyAppropriate(int phraseID, bool formal, bool religious)
        {
            // Check contextual appropriateness
            if (religious && phraseID >= 5000 && phraseID < 5100) return true;
            if (formal && phraseID >= 5100 && phraseID < 5200) return true;
            if (!formal && !religious && phraseID >= 5200 && phraseID < 5300) return
true;
            return false;
        }
    }

    public static LocalizationSystem Instance { get; private set; }

    [Header("Language Settings")]
    public string currentLanguage = "dv"; // Dhivehi language code
    public bool enableDhivehi = true;
    public bool enableFallback = true;
    public bool culturalSensitivityMode = true;

    [Header("Dhivehi Support")]
    public bool enableRTLSupport = true;
    public bool traditionalFontRendering = true;
    public bool culturalContextAwareness = true;
    public bool pronunciationGuides = true;

    [Header("Localization Features")]
```

```
public bool dynamicTextResizing = true;
public bool contextualTranslation = true;
public bool culturalAuthenticityChecking = true;
public bool respectfulLanguageMode = true;

private string[] dhivehiStrings;
private string[] englishFallback;
private bool[] culturalFlags;
private int[] authenticityScores;

// Dhivehi text database
private readonly string[] dhivehiDatabase = new string[]
{
    // Prayer times
    "ފަތުރު", // Fathuru (Dawn)
    "ހުޑުރު", // Huda (Noon)
    "ހަސްރު", // Asuru (Afternoon)
    "ހަގުރިބު", // Magurib (Sunset)
    "ޢިޝާ", // Isha (Night)

    // Common phrases
    "ައްސަލާމު ޢަލައިކުމް", // Assalaamu Alaikum (Peace be upon you)
    "شُكُرِيَا", // Shukuriyaa (Thank you)
    "ފާފުކުރޭ", // Maafu kurey (Excuse me)
    "ޢިލްތިޚާބް", // Ilthikhaab (Congratulations)

    // Island names (traditional)
```

```
    "ދަތް", // Male
    "ހުޅުދުއް", // Hulhumaale
    "ވިލިނގިލި", // Villingili
    "ހުޅުލެ", // Hulhule
    "މާފުށި" // Maafushi
};

void Awake()
{
    Instance = this;
    InitializeLocalizationSystem();
}

void InitializeLocalizationSystem()
{
    // Initialize localization arrays
    int stringCount = 1000;
    dhivehiStrings = new string[stringCount];
    englishFallback = new string[stringCount];
    culturalFlags = new bool[stringCount];
    authenticityScores = new int[stringCount];

    // Initialize Dhivehi text processing
    int textCount = 50;
    var textIDs = new NativeArray<int>(textCount, Allocator.TempJob);
    var isCultural = new NativeArray<bool>(textCount, Allocator.TempJob);
    var culturalLevels = new NativeArray<int>(textCount, Allocator.TempJob);
    var layoutData = new NativeArray<float3>(textCount, Allocator.TempJob);
    var rtlFlags = new NativeArray<bool>(textCount, Allocator.TempJob);
```

```
var fontSizes = new NativeArray<int>(textCount, Allocator.TempJob);

// Initialize with sample data
for (int i = 0; i < textCount; i++)
{
    textIDs[i] = 1000 + i; // Dhivehi text ID range
    isCultural[i] = i % 3 == 0; // Every third text is cultural
    culturalLevels[i] = i % 4; // Different cultural importance levels
}

var processingJob = new DhivehiTextProcessing
{
    textIDs = textIDs,
    isCulturalText = isCultural,
    culturalContextLevels = culturalLevels,
    textLayoutData = layoutData,
    requiresRTL = rtlFlags,
    fontSizeAdjustments = fontSizes
};

// Initialize cultural localization validation
var prayerNames = new NativeArray<int>(5, Allocator.TempJob);
var islandNames = new NativeArray<int>(10, Allocator.TempJob);
var fishingTerms = new NativeArray<int>(15, Allocator.TempJob);
var traditionalTitles = new NativeArray<int>(8, Allocator.TempJob);
var respectfulPhrases = new NativeArray<int>(12, Allocator.TempJob);
var culturalAuth = new NativeArray<bool>(50, Allocator.TempJob);
var pronunciation = new NativeArray<float>(50, Allocator.TempJob);
var contextAppropriate = new NativeArray<int>(50, Allocator.TempJob);

// Initialize cultural data
```

```
        for (int i = 0; i < 5; i++) prayerNames[i] = 1000 + i;
        for (int i = 0; i < 10; i++) islandNames[i] = 2000 + i;
        for (int i = 0; i < 15; i++) fishingTerms[i] = 3000 + i;
        for (int i = 0; i < 8; i++) traditionalTitles[i] = 4000 + i;
        for (int i = 0; i < 12; i++) respectfulPhrases[i] = 5000 + i;

        var culturalJob = new MaldivianCulturalLocalization
        {
            prayerTimeNames = prayerNames,
            islandNames = islandNames,
            fishingTerms = fishingTerms,
            traditionalTitles = traditionalTitles,
            respectfulPhrases = respectfulPhrases,
            culturalAuthenticity = culturalAuth,
            pronunciationAccuracy = pronunciation,
            contextualAppropriateness = contextAppropriate,
            currentPrayerTime = PrayerTimeSystem.Instance != null ?
PrayerTimeSystem.Instance.GetCurrentPrayerIndex() : 0,
            currentIsland = 0,
            isFormalContext = true,
            isReligiousContext = false
        };

        JobHandle processingHandle = processingJob.Schedule(textCount, 8);
        JobHandle culturalHandle = culturalJob.Schedule(processingHandle);
        culturalHandle.Complete();

        // Process results
        ProcessTextLayout(layoutData, rtlFlags, fontSizes);
        ProcessCulturalValidation(culturalAuth, pronunciation, contextAppropriate);
```

```
        // Cleanup
        textIDs.Dispose();
        isCultural.Dispose();
        culturalLevels.Dispose();
        layoutData.Dispose();
        rtlFlags.Dispose();
        fontSizes.Dispose();
        prayerNames.Dispose();
        islandNames.Dispose();
        fishingTerms.Dispose();
        traditionalTitles.Dispose();
        respectfulPhrases.Dispose();
        culturalAuth.Dispose();
        pronunciation.Dispose();
        contextAppropriate.Dispose();

        StartLocalizationMonitoring();
    }

    void ProcessTextLayout(NativeArray<float3> layouts, NativeArray<bool> rtl,
NativeArray<int> fontSizes)
    {
        // Process layout data for UI system
        for (int i = 0; i < layouts.Length; i++)
        {
            Debug.Log($"Text {i}: RTL={rtl[i]}, Layout={layouts[i]},
FontSize={fontSizes[i]}");
        }
    }
```

```
    void ProcessCulturalValidation(NativeArray<bool> authenticity,
NativeArray<float> pronunciation, NativeArray<int> appropriateness)
    {
        // Store cultural validation results
        for (int i = 0; i < authenticity.Length; i++)
        {
            culturalFlags[i] = authenticity[i];
            authenticityScores[i] = appropriateness[i];
        }
    }

    void StartLocalizationMonitoring()
    {
        // Monitor localization usage
        InvokeRepeating("UpdateLocalizationStats", 30.0f, 120.0f);
        InvokeRepeating("CheckCulturalAuthenticity", 60.0f, 300.0f);
    }

    public string GetLocalizedString(string key, bool culturalContext = false)
    {
        int stringID = GetStringID(key);

        if (currentLanguage == "dv" && enableDhivehi)
        {
            string dhivehiText = GetDhivehiString(stringID);

            if (!string.IsNullOrEmpty(dhivehiText))
            {
                return ApplyCulturalFormatting(dhivehiText, culturalContext);
            }
        }
```

```
    // Fallback to English
    if (enableFallback)
    {
        return GetEnglishString(stringID);
    }

    return key; // Return key if no translation found
}

string GetDhivehiString(int stringID)
{
    // Return Dhivehi string from database
    if (stringID >= 0 && stringID < dhivehiDatabase.Length)
    {
        return dhivehiDatabase[stringID];
    }

    // Generate placeholder Dhivehi text for development
    return GeneratePlaceholderDhivehi(stringID);
}

string GeneratePlaceholderDhivehi(int stringID)
{
    // Generate culturally appropriate placeholder text
    string[] placeholderTexts = new string[]
    {
        "ހިތަކަށް ވެއްޒެވެ", // "This is correct"
        "ޝުކުރިއްޔާ", // "Thank you"
        "ފަހަރުވާ", // "Excuse me"
```

```
        "ﺩِﺭِﺟْﻮِﺯَﻫْ", // "Congratulations"
        "ﻫَﺪَﻫْﺴَﻮﺭَّﺣْ ﻫَﺮَﺩِﺑَﺎﺣْ", // "Peace be upon you"
    };

    return placeholderTexts[stringID % placeholderTexts.Length];
}

string GetEnglishString(int stringID)
{
    // Return English fallback string
    string[] englishFallbacks = new string[]
    {
        "Hello",
        "Thank you",
        "Excuse me",
        "Congratulations",
        "Peace be upon you",
    };

    return englishFallbacks[stringID % englishFallbacks.Length];
}

string ApplyCulturalFormatting(string text, bool culturalContext)
{
    // Apply cultural formatting rules
    if (culturalContext && culturalSensitivityMode)
    {
        // Add respectful formatting for cultural content
        return $"ﻭِﺑِﺮِﻳَﺪِ: {text}"; // "Respectfully: [text]"
    }
```

```
        return text;
    }

    int GetStringID(string key)
    {
        // Simple hash-based ID generation
        int hash = 0;
        foreach (char c in key)
        {
            hash = (hash * 31 + c) % 1000;
        }
        return Mathf.Abs(hash);
    }

    public string GetPrayerTimeName(int prayerIndex)
    {
        string[] prayerNames = new string[]
        {
            GetLocalizedString("prayer_fathuru", true),  // Dawn
            GetLocalizedString("prayer_huda", true),     // Noon
            GetLocalizedString("prayer_asuru", true),    // Afternoon
            GetLocalizedString("prayer_magurib", true),  // Sunset
            GetLocalizedString("prayer_isha", true)      // Night
        };

        return prayerIndex >= 0 && prayerIndex < prayerNames.Length ?
prayerNames[prayerIndex] : "";
    }

    public string GetTraditionalIslandName(int islandID)
```

```
    {
        // Return traditional Dhivehi island names
        string[] islandNames = new string[]
        {
            GetLocalizedString("island_male", true),
            GetLocalizedString("island_hulhumaale", true),
            GetLocalizedString("island_villingili", true),
            GetLocalizedString("island_hulhule", true),
            GetLocalizedString("island_maafushi", true),
        };

        return islandID >= 0 && islandID < islandNames.Length ?
islandNames[islandID] : "";
    }

    public string GetRespectfulGreeting(bool isFormal, bool isReligious)
    {
        if (isReligious)
        {
            return GetLocalizedString("greeting_religious", true); // "ތަމްސަވްވުރު ތަރަވެހެއް"
        }
        else if (isFormal)
        {
            return GetLocalizedString("greeting_formal", true);   // Formal greeting
        }
        else
        {
            return GetLocalizedString("greeting_casual", false);  // Casual greeting
        }
    }
```

```
    public string GetFishingTerm(string term, bool traditional)
    {
        if (traditional)
        {
            return GetLocalizedString($"fishing_traditional_{term}", true);
        }
        else
        {
            return GetLocalizedString($"fishing_modern_{term}", false);
        }
    }

    public string GetCulturalInstruction(string instructionType)
    {
        // Get culturally appropriate instructions
        return instructionType switch
        {
            "prayer" => GetLocalizedString("instruction_prayer", true),
            "fishing" => GetLocalizedString("instruction_fishing", true),
            "navigation" => GetLocalizedString("instruction_navigation", true),
            "community" => GetLocalizedString("instruction_community", true),
            _ => GetLocalizedString("instruction_general", false)
        };
    }

    void UpdateLocalizationStats()
    {
        // Update localization usage statistics
        Debug.Log($"Localization active: {currentLanguage}, Dhivehi enabled:
{enableDhivehi}");
    }
```

```
void CheckCulturalAuthenticity()
{
    if (!culturalAuthenticityChecking) return;

    // Validate cultural authenticity of translations
    float authenticityScore = CalculateAuthenticityScore();
    Debug.Log($"Cultural authenticity score: {authenticityScore:P}");
}

float CalculateAuthenticityScore()
{
    // Calculate overall cultural authenticity
    int totalScores = 0;
    int validScores = 0;

    for (int i = 0; i < authenticityScores.Length; i++)
    {
        if (authenticityScores[i] > 0)
        {
            totalScores += authenticityScores[i];
            validScores++;
        }
    }

    return validScores > 0 ? (float)totalScores / (validScores * 10) : 0.0f;
}

public LocalizationSnapshot GetLocalizationSnapshot()
{
    return new LocalizationSnapshot
```

```
    {
        current_language = currentLanguage,
        dhivehi_enabled = enableDhivehi,
        rtl_support = enableRTLSupport,
        cultural_mode = culturalSensitivityMode,
        authenticity_score = CalculateAuthenticityScore(),
        total_strings = dhivehiStrings.Length,
        cultural_strings = CountCulturalStrings()
    };
}

int CountCulturalStrings()
{
    int count = 0;
    for (int i = 0; i < culturalFlags.Length; i++)
    {
        if (culturalFlags[i]) count++;
    }
    return count;
}

[System.Serializable]
public class LocalizationSnapshot
{
    public string current_language;
    public bool dhivehi_enabled;
    public bool rtl_support;
    public bool cultural_mode;
    public float authenticity_score;
    public int total_strings;
    public int cultural_strings;
```

```
    }
}
```

41. AccessibilitySystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using UnityEngine.UI;
using System.Collections.Generic;

[BurstCompile]
public class AccessibilitySystem : MonoBehaviour
{
    [BurstCompile]
    struct ScreenReaderOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> uiElementPositions;
        [ReadOnly] public NativeArray<float2> uiElementSizes;
        [ReadOnly] public NativeArray<int> elementTypes;
        [ReadOnly] public NativeArray<bool> isInteractive;
        [WriteOnly] public NativeArray<float> accessibilityScores;
        [WriteOnly] public NativeArray<bool> screenReaderPriority;
        [WriteOnly] public NativeArray<int> navigationOrder;

        public float3 screenDimensions;
        public float screenReaderFocusDistance;
        public int currentFocusElement;
```

```
    public void Execute(int index)
    {
        float3 position = uiElementPositions[index];
        float2 size = uiElementSizes[index];
        int type = elementTypes[index];
        bool interactive = isInteractive[index];

        // Calculate accessibility score
        float score = CalculateAccessibilityScore(position, size, type, interactive);
        bool priority = DetermineScreenReaderPriority(index, score);
        int navOrder = CalculateNavigationOrder(position, interactive);

        accessibilityScores[index] = score;
        screenReaderPriority[index] = priority;
        navigationOrder[index] = navOrder;
    }

    [BurstCompile]
    float CalculateAccessibilityScore(float3 position, float2 size, int type, bool
interactive)
    {
        float baseScore = 0.5f;

        // Interactive elements get higher priority
        if (interactive) baseScore += 0.3f;

        // Size-based scoring (larger elements are more accessible)
        float area = size.x * size.y;
        float sizeScore = math.saturate(area / (screenDimensions.x *
screenDimensions.y * 0.1f));
        baseScore += sizeScore * 0.2f;
```

```
            // Position-based scoring (center elements get higher scores)
            float3 center = screenDimensions * 0.5f;
            float distanceFromCenter = math.length(position - center);
            float maxDistance = math.length(screenDimensions * 0.5f);
            float positionScore = 1.0f - math.saturate(distanceFromCenter /
maxDistance);
            baseScore += positionScore * 0.1f;

            return math.saturate(baseScore);
        }

        [BurstCompile]
        bool DetermineScreenReaderPriority(int index, float score)
        {
            // Prioritize based on score and current focus
            float distanceFromFocus = math.abs(index - currentFocusElement);
            float focusBonus = math.saturate(1.0f - (distanceFromFocus / 10.0f));

            return (score + focusBonus * 0.3f) > 0.7f;
        }

        [BurstCompile]
        int CalculateNavigationOrder(float3 position, bool interactive)
        {
            // Calculate logical navigation order (top-to-bottom, left-to-right)
            float verticalPosition = 1.0f - math.saturate(position.y /
screenDimensions.y);
            float horizontalPosition = math.saturate(position.x / screenDimensions.x);

            // Interactive elements first, then by position
```

```
            int priority = interactive ? 1000 : 2000;
            int positionOrder = Mathf.RoundToInt(verticalPosition * 100) * 10 +
Mathf.RoundToInt(horizontalPosition * 10);

            return priority + positionOrder;
        }
    }

    [BurstCompile]
    struct MaldivianCulturalAccessibility : IJob
    {
        public NativeArray<bool> prayerTimeAnnouncements;
        public NativeArray<bool> culturalContextDescriptions;
        public NativeArray<bool> traditionalPracticeGuidance;
        public NativeArray<bool> respectfulBehaviorAlerts;
        public NativeArray<bool> environmentalAwarenessPrompts;

        public NativeArray<float> culturalSensitivityScores;
        public NativeArray<int> appropriateLanguageUsage;
        public NativeArray<bool> isCulturallyAppropriate;

        public float currentPrayerTimeWeight;
        public bool isReligiousContext;
        public bool isCommunityInteraction;
        public int currentIslandID;

        public void Execute()
        {
            ValidatePrayerAccessibility();
            ValidateCulturalContextAccessibility();
            ValidateTraditionalPracticeAccessibility();
```

```
            CalculateCulturalAppropriateness();
        }

        [BurstCompile]
        void ValidatePrayerAccessibility()
        {
            // Ensure prayer times are announced respectfully
            for (int i = 0; i < prayerTimeAnnouncements.Length; i++)
            {
                bool shouldAnnounce = currentPrayerTimeWeight > 0.7f &&
isReligiousContext;
                prayerTimeAnnouncements[i] = shouldAnnounce;

                if (shouldAnnounce)
                {
                    culturalSensitivityScores[i] = 0.95f; // High cultural sensitivity
                }
            }
        }

        [BurstCompile]
        void ValidateCulturalContextAccessibility()
        {
            // Provide cultural context for accessibility features
            for (int i = 0; i < culturalContextDescriptions.Length; i++)
            {
                bool needsContext = isCommunityInteraction || currentIslandID >= 0;
                culturalContextDescriptions[i] = needsContext;

                if (needsContext)
                {
```

```
                appropriateLanguageUsage[i] = 3; // High appropriateness level
            }
        }
    }

    [BurstCompile]
    void ValidateTraditionalPracticeAccessibility()
    {
        // Guide users through traditional practices respectfully
        for (int i = 0; i < traditionalPracticeGuidance.Length; i++)
        {
            bool needsGuidance = isCommunityInteraction && currentIslandID >=
0;
            traditionalPracticeGuidance[i] = needsGuidance;

            if (needsGuidance)
            {
                isCulturallyAppropriate[i] = true;
            }
        }
    }

    [BurstCompile]
    void CalculateCulturalAppropriateness()
    {
        // Calculate overall cultural appropriateness
        for (int i = 0; i < respectfulBehaviorAlerts.Length; i++)
        {
            bool needsAlert = isReligiousContext || currentPrayerTimeWeight >
0.5f;
            respectfulBehaviorAlerts[i] = needsAlert;
```

```
            if (needsAlert)
            {
                culturalSensitivityScores[prayerTimeAnnouncements.Length + i] =
0.90f;
            }
        }
    }
}

public static AccessibilitySystem Instance { get; private set; }

[Header("Screen Reader Settings")]
public bool enableScreenReader = true;
public float screenReaderSpeed = 1.0f;
public bool enableCulturalDescriptions = true;
public bool enableAudioCues = true;
public float audioCueVolume = 0.7f;

[Header("Visual Accessibility")]
public bool enableHighContrast = false;
public float textSizeMultiplier = 1.0f;
public bool enableColorBlindMode = false;
public ColorBlindType colorBlindType = ColorBlindType.None;

[Header("Motor Accessibility")]
public bool enableLargeTouchTargets = true;
public float touchTargetScale = 1.5f;
public bool enableGestureAlternatives = true;
public float inputDelayTolerance = 0.5f;
```

```
[Header("Cultural Accessibility")]
public bool respectPrayerTimeQuiet = true;
public bool useRespectfulLanguage = true;
public bool explainCulturalContext = true;
public bool guideTraditionalPractices = true;

[Header("Audio Accessibility")]
public bool enableAudioDescriptions = true;
public bool enableSubtitles = true;
public float subtitleSize = 1.2f;
public bool subtitleBackground = true;

private AudioSource screenReaderAudio;
private Queue<string> speechQueue;
private bool isSpeaking = false;
private int currentFocusElement = 0;
private List<AccessibleElement> accessibleElements;
private Dictionary<string, AudioClip> audioCues;

public enum ColorBlindType
{
    None,
    Protanopia,
    Deuteranopia,
    Tritanopia,
    Achromatopsia
}

void Awake()
{
    Instance = this;
```

```
        InitializeAccessibilitySystem();
    }

    void InitializeAccessibilitySystem()
    {
        // Initialize audio source for screen reader
        screenReaderAudio = gameObject.AddComponent<AudioSource>();
        screenReaderAudio.playOnAwake = false;
        screenReaderAudio.volume = audioCueVolume;

        // Initialize collections
        speechQueue = new Queue<string>();
        accessibleElements = new List<AccessibleElement>();
        audioCues = new Dictionary<string, AudioClip>();

        // Initialize accessibility calculations
        int elementCount = 50;
        var positions = new NativeArray<float3>(elementCount, Allocator.TempJob);
        var sizes = new NativeArray<float2>(elementCount, Allocator.TempJob);
        var types = new NativeArray<int>(elementCount, Allocator.TempJob);
        var interactive = new NativeArray<bool>(elementCount,
Allocator.TempJob);
        var scores = new NativeArray<float>(elementCount, Allocator.TempJob);
        var priorities = new NativeArray<bool>(elementCount, Allocator.TempJob);
        var navOrder = new NativeArray<int>(elementCount, Allocator.TempJob);

        // Initialize with sample UI data
        for (int i = 0; i < elementCount; i++)
        {
            positions[i] = UnityEngine.Random.insideUnitSphere * 10.0f;
            sizes[i] = UnityEngine.Random.Range(0.5f, 3.0f) * Vector2.one;
```

```
    types[i] = UnityEngine.Random.Range(0, 5);
    interactive[i] = UnityEngine.Random.Range(0, 2) == 1;
}

var optimizationJob = new ScreenReaderOptimization
{
    uiElementPositions = positions,
    uiElementSizes = sizes,
    elementTypes = types,
    isInteractive = interactive,
    accessibilityScores = scores,
    screenReaderPriority = priorities,
    navigationOrder = navOrder,
    screenDimensions = new float3(Screen.width, Screen.height, 0),
    screenReaderFocusDistance = 5.0f,
    currentFocusElement = currentFocusElement
};

// Initialize cultural accessibility
var prayerAnnouncements = new NativeArray<bool>(5, Allocator.TempJob);
var culturalDescriptions = new NativeArray<bool>(10, Allocator.TempJob);
var traditionalGuidance = new NativeArray<bool>(8, Allocator.TempJob);
var behaviorAlerts = new NativeArray<bool>(12, Allocator.TempJob);
var environmentalPrompts = new NativeArray<bool>(6, Allocator.TempJob);
var sensitivityScores = new NativeArray<float>(50, Allocator.TempJob);
var languageUsage = new NativeArray<int>(50, Allocator.TempJob);
var culturalAppropriate = new NativeArray<bool>(50, Allocator.TempJob);

var culturalJob = new MaldivianCulturalAccessibility
{
    prayerTimeAnnouncements = prayerAnnouncements,
```

```
            culturalContextDescriptions = culturalDescriptions,
            traditionalPracticeGuidance = traditionalGuidance,
            respectfulBehaviorAlerts = behaviorAlerts,
            environmentalAwarenessPrompts = environmentalPrompts,
            culturalSensitivityScores = sensitivityScores,
            appropriateLanguageUsage = languageUsage,
            isCulturallyAppropriate = culturalAppropriate,
            currentPrayerTimeWeight = GetPrayerTimeWeight(),
            isReligiousContext = IsReligiousContext(),
            isCommunityInteraction = IsCommunityInteraction(),
            currentIslandID = GetCurrentIslandID()
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(elementCount,
8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessAccessibilityScores(scores, priorities, navOrder);
        ProcessCulturalAccessibility(prayerAnnouncements, culturalDescriptions,
traditionalGuidance, sensitivityScores);

        // Cleanup
        positions.Dispose();
        sizes.Dispose();
        types.Dispose();
        interactive.Dispose();
        scores.Dispose();
        priorities.Dispose();
        navOrder.Dispose();
```

```
        prayerAnnouncements.Dispose();
        culturalDescriptions.Dispose();
        traditionalGuidance.Dispose();
        behaviorAlerts.Dispose();
        environmentalPrompts.Dispose();
        sensitivityScores.Dispose();
        languageUsage.Dispose();
        culturalAppropriate.Dispose();

        RegisterAccessibleElements();
        StartAccessibilityMonitoring();
    }

    void ProcessAccessibilityScores(NativeArray<float> scores,
NativeArray<bool> priorities, NativeArray<int> navOrder)
    {
        // Process accessibility score results
        int highPriorityCount = 0;
        for (int i = 0; i < priorities.Length; i++)
        {
            if (priorities[i]) highPriorityCount++;
        }
        Debug.Log($"High priority screen reader elements:
{highPriorityCount}/{priorities.Length}");
    }

    void ProcessCulturalAccessibility(NativeArray<bool> prayers,
NativeArray<bool> cultural, NativeArray<bool> traditional, NativeArray<float>
sensitivity)
    {
        // Process cultural accessibility requirements
```

```
        int culturalElements = 0;
        for (int i = 0; i < cultural.Length; i++)
        {
            if (cultural[i]) culturalElements++;
        }
        Debug.Log($"Cultural accessibility elements: {culturalElements}");
    }

    void RegisterAccessibleElements()
    {
        // Register UI elements for accessibility
        AccessibleElement[] elements = FindObjectsOfType<AccessibleElement>();
        accessibleElements.AddRange(elements);

        Debug.Log($"Registered {accessibleElements.Count} accessible
elements");
    }

    void StartAccessibilityMonitoring()
    {
        // Monitor accessibility needs
        InvokeRepeating("UpdateScreenReaderFocus", 0.5f, 1.0f);
        InvokeRepeating("CheckCulturalAccessibility", 1.0f, 5.0f);
    }

    public void SpeakText(string text, bool interrupt = false)
    {
        if (!enableScreenReader) return;

        if (interrupt)
        {
```

```
        StopSpeaking();
    }

    speechQueue.Enqueue(text);

    if (!isSpeaking)
    {
        StartCoroutine(ProcessSpeechQueue());
    }
}

System.Collections.IEnumerator ProcessSpeechQueue()
{
    isSpeaking = true;

    while (speechQueue.Count > 0)
    {
        string text = speechQueue.Dequeue();

        // Apply cultural sensitivity
        if (respectPrayerTimeQuiet && IsPrayerTime())
        {
            text = ModifyForPrayerTime(text);
        }

        // Simulate text-to-speech (in production, use native TTS)
        Debug.Log($"Screen Reader: {text}");

        // Calculate speech duration
        float duration = text.Length / (screenReaderSpeed * 10.0f);
```

```
            yield return new WaitForSeconds(duration);
        }

        isSpeaking = false;
    }

    string ModifyForPrayerTime(string text)
    {
        // Modify speech during prayer times for respect
        return $"[Quiet mode - Prayer time] {text}";
    }

    public void StopSpeaking()
    {
        speechQueue.Clear();
        isSpeaking = false;
        StopCoroutine(ProcessSpeechQueue());
    }

    public void AnnounceUIElement(string elementName, string description, string
elementType)
    {
        if (!enableScreenReader) return;

        string announcement = $"{elementName}. {elementType}. {description}";

        if (explainCulturalContext && IsCulturalElement(elementType))
        {
            announcement += GetCulturalContext(elementType);
        }
```

```
        SpeakText(announcement, true);
    }

    bool IsCulturalElement(string elementType)
    {
        return elementType.Contains("Prayer") ||
elementType.Contains("Traditional") ||
               elementType.Contains("Cultural") || elementType.Contains("Religious");
    }

    string GetCulturalContext(string elementType)
    {
        return elementType switch
        {
            "PrayerButton" => " This is a prayer time announcement. Please observe
respectfully.",
            "TraditionalFishing" => " This represents traditional Maldivian fishing
practices.",
            "CulturalEvent" => " This is an important cultural event in Maldivian
tradition.",
            _ => ""
        };
    }

    public void ApplyHighContrastMode()
    {
        if (!enableHighContrast) return;

        // Apply high contrast to UI elements
        foreach (var graphic in FindObjectsOfType<Graphic>())
        {
```

```
            Color highContrastColor = GetHighContrastColor(graphic.color);
            graphic.color = highContrastColor;
        }
    }

    Color GetHighContrastColor(Color original)
    {
        // Convert to high contrast
        float gray = original.grayscale;
        return gray > 0.5f ? Color.white : Color.black;
    }

    public void ApplyColorBlindMode()
    {
        if (!enableColorBlindMode) return;

        // Apply color blind friendly palette
        foreach (var graphic in FindObjectsOfType<Graphic>())
        {
            Color adjustedColor = AdjustForColorBlindness(graphic.color);
            graphic.color = adjustedColor;
        }
    }

    Color AdjustForColorBlindness(Color color)
    {
        return colorBlindType switch
        {
            ColorBlindType.Protanopia => AdjustForProtanopia(color),
            ColorBlindType.Deuteranopia => AdjustForDeuteranopia(color),
            ColorBlindType.Tritanopia => AdjustForTritanopia(color),
```

```
        ColorBlindType.Achromatopsia => Color.grayscale,
        _ => color
    };
}

Color AdjustForProtanopia(Color color)
{
    // Simulate protanopia (red-blind)
    return new Color(color.g * 0.5f + color.b * 0.5f, color.g, color.b, color.a);
}

Color AdjustForDeuteranopia(Color color)
{
    // Simulate deuteranopia (green-blind)
    return new Color(color.r, color.r * 0.5f + color.b * 0.5f, color.b, color.a);
}

Color AdjustForTritanopia(Color color)
{
    // Simulate tritanopia (blue-blind)
    return new Color(color.r, color.g, color.r * 0.5f + color.g * 0.5f, color.a);
}

float GetPrayerTimeWeight()
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0.0f;
}
```

```
bool IsPrayerTime()
{
    return GetPrayerTimeWeight() > 0.7f;
}

bool IsReligiousContext()
{
    // Check if player is in religious context
    return GetPrayerTimeWeight() > 0.5f;
}

bool IsCommunityInteraction()
{
    // Check if player is interacting with community
    return Time.time % 86400 > 43200; // Simplified check
}

int GetCurrentIslandID()
{
    // Get current island ID
    return 0; // Placeholder
}

public AccessibilitySnapshot GetAccessibilitySnapshot()
{
    return new AccessibilitySnapshot
    {
        screen_reader_enabled = enableScreenReader,
        high_contrast_active = enableHighContrast,
        color_blind_mode = enableColorBlindMode,
```

```
            large_touch_targets = enableLargeTouchTargets,
            cultural_accessibility = respectPrayerTimeQuiet,
            elements_registered = accessibleElements.Count,
            speech_queue_size = speechQueue.Count,
            cultural_sensitivity_active = IsPrayerTime()
        };
    }

    [System.Serializable]
    public class AccessibilitySnapshot
    {
        public bool screen_reader_enabled;
        public bool high_contrast_active;
        public bool color_blind_mode;
        public bool large_touch_targets;
        public bool cultural_accessibility;
        public int elements_registered;
        public int speech_queue_size;
        public bool cultural_sensitivity_active;
    }
}
```

42. DebugSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;
```

```
using System.Text;

[BurstCompile]
public class DebugSystem : MonoBehaviour
{
    [BurstCompile]
    struct PerformanceMetricsAnalysis : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> frameTimeHistory;
        [ReadOnly] public NativeArray<float> memoryUsageHistory;
        [ReadOnly] public NativeArray<int> drawCallHistory;
        [ReadOnly] public NativeArray<float> gpuTimeHistory;
        [WriteOnly] public NativeArray<float> performanceScores;
        [WriteOnly] public NativeArray<bool> performanceWarnings;
        [WriteOnly] public NativeArray<int> optimizationPriorities;

        public float targetFrameTime;
        public float maxMemoryUsage;
        public int maxDrawCalls;
        public float maxGPUTime;

        public void Execute(int index)
        {
            float frameTime = frameTimeHistory[index];
            float memoryUsage = memoryUsageHistory[index];
            int drawCalls = drawCallHistory[index];
            float gpuTime = gpuTimeHistory[index];

            // Calculate performance score
            float score = CalculatePerformanceScore(frameTime, memoryUsage,
drawCalls, gpuTime);
```

```
            bool warning = ShouldWarnAboutPerformance(score);
            int priority = CalculateOptimizationPriority(score, frameTime,
memoryUsage);

            performanceScores[index] = score;
            performanceWarnings[index] = warning;
            optimizationPriorities[index] = priority;
        }

        [BurstCompile]
        float CalculatePerformanceScore(float frameTime, float memoryUsage, int
drawCalls, float gpuTime)
        {
            float frameScore = math.saturate(1.0f - (frameTime / (targetFrameTime *
2.0f)));
            float memoryScore = math.saturate(1.0f - (memoryUsage /
maxMemoryUsage));
            float drawCallScore = math.saturate(1.0f - ((float)drawCalls /
(float)maxDrawCalls));
            float gpuScore = math.saturate(1.0f - (gpuTime / maxGPUTime));

            return (frameScore + memoryScore + drawCallScore + gpuScore) * 0.25f;
        }

        [BurstCompile]
        bool ShouldWarnAboutPerformance(float score)
        {
            return score < 0.7f; // Warn if performance score is below 70%
        }

        [BurstCompile]
```

```
    int CalculateOptimizationPriority(float score, float frameTime, float
memoryUsage)
    {
        int priority = 0;

        if (frameTime > targetFrameTime * 1.5f) priority += 3;
        if (memoryUsage > maxMemoryUsage * 0.8f) priority += 2;
        if (score < 0.5f) priority += 1;

        return math.min(priority, 5);
    }
}

[BurstCompile]
struct MaldivianCulturalDebug : IJob
{
    public NativeArray<bool> culturalAccuracyChecks;
    public NativeArray<bool> respectfulContentValidation;
    public NativeArray<bool> prayerTimeAccuracy;
    public NativeArray<bool> traditionalPracticeAuthenticity;
    public NativeArray<bool> languageAppropriateness;

    public NativeArray<float> culturalSensitivityScores;
    public NativeArray<int> authenticityViolations;
    public NativeArray<string> culturalWarnings;

    public int currentPrayerTimeIndex;
    public float prayerAccuracy;
    public bool isReligiousContext;
    public int currentIslandID;
    public string currentLocale;
```

```
    public void Execute()
    {
        ValidateCulturalAccuracy();
        ValidateRespectfulContent();
        ValidatePrayerTimeAccuracy();
        ValidateTraditionalPractices();
        CalculateCulturalSensitivity();
    }

    [BurstCompile]
    void ValidateCulturalAccuracy()
    {
        // Validate cultural content accuracy
        for (int i = 0; i < culturalAccuracyChecks.Length; i++)
        {
            bool isAccurate = ValidateIslandCulture(i) &&
ValidateTraditionalPractices(i);
            culturalAccuracyChecks[i] = isAccurate;

            if (!isAccurate)
            {
                authenticityViolations[i]++;
            }
        }
    }

    [BurstCompile]
    void ValidateRespectfulContent()
    {
        // Validate content respects Maldivian culture
```

```
        for (int i = 0; i < respectfulContentValidation.Length; i++)
        {
            bool isRespectful = !ContainsDisrespectfulContent(i) &&
RespectsReligiousSensitivities(i);
            respectfulContentValidation[i] = isRespectful;

            if (!isRespectful)
            {
                culturalWarnings[i] = "Content may be culturally insensitive";
            }
        }
    }

    [BurstCompile]
    void ValidatePrayerTimeAccuracy()
    {
        // Validate prayer time calculations
        for (int i = 0; i < prayerTimeAccuracy.Length; i++)
        {
            bool isAccurate = math.abs(prayerAccuracy - 1.0f) < 0.1f;
            prayerTimeAccuracy[i] = isAccurate;

            if (!isAccurate)
            {
                culturalWarnings[prayerTimeAccuracy.Length + i] = "Prayer time
calculation may be inaccurate";
            }
        }
    }

    [BurstCompile]
```

```
    void ValidateTraditionalPractices()
    {
        // Validate traditional practice representations
        for (int i = 0; i < traditionalPracticeAuthenticity.Length; i++)
        {
            bool isAuthentic = ValidateFishingPractices(i) &&
ValidateNavigationMethods(i);
            traditionalPracticeAuthenticity[i] = isAuthentic;

            if (isAuthentic)
            {
                culturalSensitivityScores[i] = 0.95f;
            }
        }
    }

    [BurstCompile]
    void CalculateCulturalSensitivity()
    {
        // Calculate overall cultural sensitivity score
        float totalScore = 0.0f;
        int totalChecks = 0;

        for (int i = 0; i < culturalAccuracyChecks.Length; i++)
        {
            if (culturalAccuracyChecks[i]) totalScore += 1.0f;
            totalChecks++;
        }

        for (int i = 0; i < respectfulContentValidation.Length; i++)
        {
```

```
            if (respectfulContentValidation[i]) totalScore += 1.0f;
            totalChecks++;
        }

        float averageScore = totalChecks > 0 ? totalScore / totalChecks : 0.0f;

        // Store overall sensitivity score
        for (int i = 0; i < culturalSensitivityScores.Length; i++)
        {
            culturalSensitivityScores[i] = averageScore;
        }
    }

    [BurstCompile]
    bool ValidateIslandCulture(int index)
    {
        // Validate island-specific cultural accuracy
        return currentIslandID >= 0 && currentIslandID < 41; // Valid Maldivian
island range
    }

    [BurstCompile]
    bool ValidateTraditionalPractices(int index)
    {
        // Validate traditional practice accuracy
        return currentLocale == "dv" || currentLocale == "en"; // Only support
Dhivehi and English
    }

    [BurstCompile]
    bool ContainsDisrespectfulContent(int index)
```

```
        {
            // Check for culturally disrespectful content
            return false; // Simplified - would check against cultural database
        }

        [BurstCompile]
        bool RespectsReligiousSensitivities(int index)
        {
            // Check if content respects religious sensitivities
            return !isReligiousContext || currentPrayerTimeIndex < 0 ||
currentPrayerTimeIndex >= 5;
        }

        [BurstCompile]
        bool ValidateFishingPractices(int index)
        {
            // Validate traditional fishing practice representations
            return true; // Simplified validation
        }

        [BurstCompile]
        bool ValidateNavigationMethods(int index)
        {
            // Validate traditional navigation method representations
            return true; // Simplified validation
        }
    }

    public static DebugSystem Instance { get; private set; }

    [Header("Debug Settings")]
```

```
public bool enableDebugConsole = true;
public bool enablePerformanceMonitoring = true;
public bool enableCulturalValidation = true;
public bool enableMemoryTracking = true;
public KeyCode consoleToggleKey = KeyCode.BackQuote;

[Header("Performance Monitoring")]
public bool showFPS = true;
public bool showMemoryUsage = true;
public bool showDrawCalls = true;
public bool showGPUTime = true;
public float performanceUpdateInterval = 0.5f;

[Header("Cultural Debug")]
public bool validateCulturalAccuracy = true;
public bool checkRespectfulContent = true;
public bool monitorPrayerTimes = true;
public bool trackTraditionalPractices = true;

[Header("Debug Console")]
public int maxConsoleLines = 100;
public bool enableCommandInput = true;
public bool showTimestamp = true;
public Color consoleBackgroundColor = new Color(0, 0, 0, 0.8f);
public Color consoleTextColor = Color.green;

private List<string> consoleOutput;
private bool showConsole = false;
private string commandInput = "";
private Vector2 consoleScrollPosition;
private float lastPerformanceUpdate;
```

```
    private float[] frameTimeHistory;
    private float[] memoryUsageHistory;
    private int[] drawCallHistory;
    private float[] gpuTimeHistory;
    private int historyIndex = 0;

    void Awake()
    {
        Instance = this;
        InitializeDebugSystem();
    }

    void InitializeDebugSystem()
    {
        // Initialize collections
        consoleOutput = new List<string>();
        frameTimeHistory = new float[60];
        memoryUsageHistory = new float[60];
        drawCallHistory = new int[60];
        gpuTimeHistory = new float[60];

        // Initialize performance analysis
        int sampleCount = 60;
        var frameTimes = new NativeArray<float>(sampleCount,
Allocator.TempJob);
        var memoryUsages = new NativeArray<float>(sampleCount,
Allocator.TempJob);
        var drawCalls = new NativeArray<int>(sampleCount, Allocator.TempJob);
        var gpuTimes = new NativeArray<float>(sampleCount, Allocator.TempJob);
        var scores = new NativeArray<float>(sampleCount, Allocator.TempJob);
        var warnings = new NativeArray<bool>(sampleCount, Allocator.TempJob);
```

```
        var priorities = new NativeArray<int>(sampleCount, Allocator.TempJob);

        // Initialize with sample data
        for (int i = 0; i < sampleCount; i++)
        {
            frameTimes[i] = UnityEngine.Random.Range(0.008f, 0.02f); // 50-125
FPS
            memoryUsages[i] = UnityEngine.Random.Range(100f, 300f); // 100-300
MB
            drawCalls[i] = UnityEngine.Random.Range(50, 200);
            gpuTimes[i] = UnityEngine.Random.Range(0.005f, 0.015f);
        }

        var performanceJob = new PerformanceMetricsAnalysis
        {
            frameTimeHistory = frameTimes,
            memoryUsageHistory = memoryUsages,
            drawCallHistory = drawCalls,
            gpuTimeHistory = gpuTimes,
            performanceScores = scores,
            performanceWarnings = warnings,
            optimizationPriorities = priorities,
            targetFrameTime = 0.0167f, // 60 FPS
            maxMemoryUsage = 500f, // 500 MB
            maxDrawCalls = 300,
            maxGPUTime = 0.0167f
        };

        // Initialize cultural debug validation
        var culturalAccuracy = new NativeArray<bool>(20, Allocator.TempJob);
        var respectfulContent = new NativeArray<bool>(15, Allocator.TempJob);
```

```
        var prayerAccuracy = new NativeArray<bool>(5, Allocator.TempJob);
        var traditionalAuthenticity = new NativeArray<bool>(10, Allocator.TempJob);
        var languageAppropriate = new NativeArray<bool>(25, Allocator.TempJob);
        var culturalScores = new NativeArray<float>(50, Allocator.TempJob);
        var violations = new NativeArray<int>(50, Allocator.TempJob);
        var warnings = new NativeArray<string>(50, Allocator.TempJob);

        var culturalJob = new MaldivianCulturalDebug
        {
            culturalAccuracyChecks = culturalAccuracy,
            respectfulContentValidation = respectfulContent,
            prayerTimeAccuracy = prayerAccuracy,
            traditionalPracticeAuthenticity = traditionalAuthenticity,
            languageAppropriateness = languageAppropriate,
            culturalSensitivityScores = culturalScores,
            authenticityViolations = violations,
            culturalWarnings = warnings,
            currentPrayerTimeIndex = PrayerTimeSystem.Instance != null ?
PrayerTimeSystem.Instance.GetCurrentPrayerIndex() : 0,
            prayerAccuracy = 0.95f,
            isReligiousContext = false,
            currentIslandID = 0,
            currentLocale = "dv"
        };

        JobHandle performanceHandle = performanceJob.Schedule(sampleCount,
8);
        JobHandle culturalHandle = culturalJob.Schedule(performanceHandle);
        culturalHandle.Complete();

        // Process results
```

```
        ProcessPerformanceAnalysis(scores, warnings, priorities);
        ProcessCulturalValidation(culturalAccuracy, respectfulContent,
prayerAccuracy, culturalScores, violations);

        // Cleanup
        frameTimes.Dispose();
        memoryUsages.Dispose();
        drawCalls.Dispose();
        gpuTimes.Dispose();
        scores.Dispose();
        warnings.Dispose();
        priorities.Dispose();
        culturalAccuracy.Dispose();
        respectfulContent.Dispose();
        prayerAccuracy.Dispose();
        traditionalAuthenticity.Dispose();
        languageAppropriate.Dispose();
        culturalScores.Dispose();
        violations.Dispose();
        warnings.Dispose();

        StartDebugMonitoring();
        Log("Debug System initialized successfully", LogType.System);
    }

    void ProcessPerformanceAnalysis(NativeArray<float> scores,
NativeArray<bool> warnings, NativeArray<int> priorities)
    {
        // Process performance analysis results
        int warningCount = 0;
        int highPriorityCount = 0;
```

```
        for (int i = 0; i < warnings.Length; i++)
        {
            if (warnings[i]) warningCount++;
            if (priorities[i] >= 3) highPriorityCount++;
        }

        Log($"Performance analysis: {warningCount} warnings, {highPriorityCount}
high priority optimizations needed", LogType.Performance);
    }

    void ProcessCulturalValidation(NativeArray<bool> accuracy,
NativeArray<bool> respect, NativeArray<bool> prayer, NativeArray<float>
scores, NativeArray<int> violations)
    {
        // Process cultural validation results
        int accuracyIssues = 0;
        int respectIssues = 0;

        for (int i = 0; i < accuracy.Length; i++)
        {
            if (!accuracy[i]) accuracyIssues++;
        }

        for (int i = 0; i < respect.Length; i++)
        {
            if (!respect[i]) respectIssues++;
        }

        Log($"Cultural validation: {accuracyIssues} accuracy issues, {respectIssues}
respect issues", LogType.Cultural);
```

```
    }

    void StartDebugMonitoring()
    {
        // Start various monitoring systems
        InvokeRepeating("UpdatePerformanceMetrics", 0.0f, 1.0f / 60.0f); // Every
frame
        InvokeRepeating("UpdateMemoryTracking", 1.0f, 1.0f);
        InvokeRepeating("ValidateCulturalContent", 5.0f, 30.0f);
    }

    void Update()
    {
        // Toggle console
        if (Input.GetKeyDown(consoleToggleKey))
        {
            showConsole = !showConsole;
        }

        // Update performance metrics
        if (Time.time - lastPerformanceUpdate >= performanceUpdateInterval)
        {
            UpdatePerformanceMetrics();
            lastPerformanceUpdate = Time.time;
        }
    }

    void UpdatePerformanceMetrics()
    {
        // Record current performance metrics
        frameTimeHistory[historyIndex] = Time.unscaledDeltaTime;
```

```
        memoryUsageHistory[historyIndex] =
UnityEngine.Profiling.Profiler.GetTotalAllocatedMemory(false) / (1024f * 1024f); //
MB
        drawCallHistory[historyIndex] = UnityStats.drawCalls;
        gpuTimeHistory[historyIndex] = UnityStats.renderTime / 1000f; // Convert to
seconds

        historyIndex = (historyIndex + 1) % frameTimeHistory.Length;
    }

    void UpdateMemoryTracking()
    {
        if (!enableMemoryTracking) return;

        long totalMemory =
UnityEngine.Profiling.Profiler.GetTotalAllocatedMemory(false);
        long monoMemory =
UnityEngine.Profiling.Profiler.GetMonoUsedSizeLong();

        Log($"Memory - Total: {totalMemory / (1024f * 1024f):F1}MB, Mono:
{monoMemory / (1024f * 1024f):F1}MB", LogType.Memory);
    }

    void ValidateCulturalContent()
    {
        if (!enableCulturalValidation) return;

        // Perform cultural validation checks
        bool prayerAccuracy = ValidatePrayerTimes();
        bool culturalRespect = ValidateCulturalRespect();
```

```
        if (!prayerAccuracy)
        {
            Log("Warning: Prayer time calculations may be inaccurate",
LogType.Cultural);
        }

        if (!culturalRespect)
        {
            Log("Warning: Some content may lack cultural sensitivity",
LogType.Cultural);
        }
    }

    bool ValidatePrayerTimes()
    {
        // Validate prayer time accuracy
        if (PrayerTimeSystem.Instance != null)
        {
            float accuracy = PrayerTimeSystem.Instance.GetPrayerTimeAccuracy();
            return accuracy > 0.9f;
        }
        return true;
    }

    bool ValidateCulturalRespect()
    {
        // Validate cultural respect in content
        // Simplified validation
        return true;
    }
```

```
    void OnGUI()
    {
        if (!showConsole || !enableDebugConsole) return;

        DrawConsoleWindow();
    }

    void DrawConsoleWindow()
    {
        // Console background
        GUI.color = consoleBackgroundColor;
        GUI.DrawTexture(new Rect(0, 0, Screen.width, Screen.height * 0.5f),
Texture2D.whiteTexture);

        // Console content
        GUI.color = consoleTextColor;

        // Draw performance metrics
        if (showFPS)
        {
            string fpsText = $"FPS: {1.0f / Time.unscaledDeltaTime:F1}";
            GUI.Label(new Rect(10, 10, 200, 20), fpsText);
        }

        // Draw console output
        consoleScrollPosition = GUI.BeginScrollView(new Rect(10, 40,
Screen.width - 20, Screen.height * 0.4f - 50),
            consoleScrollPosition, new Rect(0, 0, Screen.width - 40,
consoleOutput.Count * 20));

        for (int i = 0; i < consoleOutput.Count; i++)
```

```
        {
            GUI.Label(new Rect(0, i * 20, Screen.width - 40, 20), consoleOutput[i]);
        }

        GUI.EndScrollView();

        // Command input
        if (enableCommandInput)
        {
            GUI.SetNextControlName("DebugCommandInput");
            commandInput = GUI.TextField(new Rect(10, Screen.height * 0.4f + 10,
Screen.width - 120, 25), commandInput);

            if (GUI.Button(new Rect(Screen.width - 100, Screen.height * 0.4f + 10,
80, 25), "Execute") ||
                (Event.current.type == EventType.KeyDown &&
Event.current.keyCode == KeyCode.Return &&
GUI.GetNameOfFocusedControl() == "DebugCommandInput"))
            {
                ExecuteCommand(commandInput);
                commandInput = "";
            }
        }
    }

    public void Log(string message, LogType logType = LogType.General)
    {
        string timestamp = showTimestamp ?
$"[{System.DateTime.Now:HH:mm:ss}] " : "";
        string typePrefix = logType switch
        {
```

```
            LogType.System => "[SYSTEM] ",
            LogType.Performance => "[PERF] ",
            LogType.Cultural => "[CULTURAL] ",
            LogType.Memory => "[MEMORY] ",
            LogType.Warning => "[WARNING] ",
            LogType.Error => "[ERROR] ",
            _ => "[INFO] "
        };

        string fullMessage = timestamp + typePrefix + message;

        consoleOutput.Add(fullMessage);

        // Keep only recent messages
        if (consoleOutput.Count > maxConsoleLines)
        {
            consoleOutput.RemoveAt(0);
        }

        // Auto-scroll to bottom
        consoleScrollPosition.y = Mathf.Infinity;

        // Also log to Unity console
        Debug.Log(fullMessage);
    }

    void ExecuteCommand(string command)
    {
        string[] parts = command.Split(' ');
        string cmd = parts[0].ToLower();
```

```
    switch (cmd)
    {
        case "help":
            ShowHelp();
            break;
        case "fps":
            showFPS = !showFPS;
            Log($"FPS display: {showFPS}");
            break;
        case "memory":
            showMemoryUsage = !showMemoryUsage;
            Log($"Memory display: {showMemoryUsage}");
            break;
        case "clear":
            consoleOutput.Clear();
            break;
        case "culture":
            ValidateCulturalContent();
            break;
        case "performance":
            LogPerformanceSummary();
            break;
        default:
            Log($"Unknown command: {cmd}", LogType.Warning);
            break;
    }
}

void ShowHelp()
{
    Log("Available commands:", LogType.System);
```

```
    Log("  help - Show this help");
    Log("  fps - Toggle FPS display");
    Log("  memory - Toggle memory display");
    Log("  clear - Clear console");
    Log("  culture - Run cultural validation");
    Log("  performance - Show performance summary");
}

void LogPerformanceSummary()
{
    float avgFrameTime = 0f;
    float avgMemory = 0f;

    for (int i = 0; i < frameTimeHistory.Length; i++)
    {
        avgFrameTime += frameTimeHistory[i];
        avgMemory += memoryUsageHistory[i];
    }

    avgFrameTime /= frameTimeHistory.Length;
    avgMemory /= memoryUsageHistory.Length;

    Log($"Performance Summary:", LogType.Performance);
    Log($"  Average FPS: {1.0f / avgFrameTime:F1}");
    Log($"  Average Memory: {avgMemory:F1}MB");
    Log($"  Current Frame Time: {Time.unscaledDeltaTime * 1000f:F1}ms");
}

public enum LogType
{
    General,
```

```
    System,
    Performance,
    Cultural,
    Memory,
    Warning,
    Error
}

public DebugSnapshot GetDebugSnapshot()
{
    return new DebugSnapshot
    {
        console_enabled = enableDebugConsole,
        performance_monitoring = enablePerformanceMonitoring,
        cultural_validation = enableCulturalValidation,
        memory_tracking = enableMemoryTracking,
        console_lines = consoleOutput.Count,
        warnings_count = CountWarnings(),
        errors_count = CountErrors(),
        average_fps = 1.0f / Time.unscaledDeltaTime
    };
}

int CountWarnings()
{
    int count = 0;
    foreach (string line in consoleOutput)
    {
        if (line.Contains("[WARNING]")) count++;
    }
    return count;
```

```
    }

    int CountErrors()
    {
        int count = 0;
        foreach (string line in consoleOutput)
        {
            if (line.Contains("[ERROR]")) count++;
        }
        return count;
    }

    [System.Serializable]
    public class DebugSnapshot
    {
        public bool console_enabled;
        public bool performance_monitoring;
        public bool cultural_validation;
        public bool memory_tracking;
        public int console_lines;
        public int warnings_count;
        public int errors_count;
        public float average_fps;
    }
}
```

43. PerformanceProfiler.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using Unity.Profiling;
using System.Collections.Generic;

[BurstCompile]
public class PerformanceProfiler : MonoBehaviour
{
    [BurstCompile]
    struct MobilePerformanceOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> cpuTimeHistory;
        [ReadOnly] public NativeArray<float> gpuTimeHistory;
        [ReadOnly] public NativeArray<int> drawCallHistory;
        [ReadOnly] public NativeArray<int> triangleCountHistory;
        [ReadOnly] public NativeArray<float> memoryUsageHistory;
        [WriteOnly] public NativeArray<float> optimizationScores;
        [WriteOnly] public NativeArray<bool> performanceBottlenecks;
        [WriteOnly] public NativeArray<int> optimizationRecommendations;

        public float targetFrameTime;
        public int maxDrawCalls;
        public int maxTriangles;
        public float maxMemoryUsage;
        public MobileTier mobileTier;

        public void Execute(int index)
        {
```

```
            float cpuTime = cpuTimeHistory[index];
            float gpuTime = gpuTimeHistory[index];
            int drawCalls = drawCallHistory[index];
            int triangles = triangleCountHistory[index];
            float memoryUsage = memoryUsageHistory[index];

            // Calculate optimization score
            float score = CalculateOptimizationScore(cpuTime, gpuTime, drawCalls,
triangles, memoryUsage);
            bool bottleneck = DetectPerformanceBottleneck(cpuTime, gpuTime,
drawCalls, memoryUsage);
            int recommendation = GenerateOptimizationRecommendation(cpuTime,
gpuTime, drawCalls, triangles, memoryUsage);

            optimizationScores[index] = score;
            performanceBottlenecks[index] = bottleneck;
            optimizationRecommendations[index] = recommendation;
        }

        [BurstCompile]
        float CalculateOptimizationScore(float cpuTime, float gpuTime, int
drawCalls, int triangles, float memoryUsage)
        {
            float cpuScore = math.saturate(1.0f - (cpuTime / targetFrameTime));
            float gpuScore = math.saturate(1.0f - (gpuTime / targetFrameTime));
            float drawCallScore = math.saturate(1.0f - ((float)drawCalls /
(float)maxDrawCalls));
            float triangleScore = math.saturate(1.0f - ((float)triangles /
(float)maxTriangles));
            float memoryScore = math.saturate(1.0f - (memoryUsage /
maxMemoryUsage));
```

```
            return (cpuScore + gpuScore + drawCallScore + triangleScore +
memoryScore) * 0.2f;
        }

        [BurstCompile]
        bool DetectPerformanceBottleneck(float cpuTime, float gpuTime, int
drawCalls, float memoryUsage)
        {
            return cpuTime > targetFrameTime * 0.7f ||
                   gpuTime > targetFrameTime * 0.7f ||
                   drawCalls > maxDrawCalls * 0.8f ||
                   memoryUsage > maxMemoryUsage * 0.8f;
        }

        [BurstCompile]
        int GenerateOptimizationRecommendation(float cpuTime, float gpuTime, int
drawCalls, int triangles, float memoryUsage)
        {
            if (cpuTime > targetFrameTime * 0.7f) return 1; // Reduce CPU load
            if (gpuTime > targetFrameTime * 0.7f) return 2; // Reduce GPU load
            if (drawCalls > maxDrawCalls * 0.8f) return 3; // Reduce draw calls
            if (triangles > maxTriangles * 0.8f) return 4; // Reduce triangle count
            if (memoryUsage > maxMemoryUsage * 0.8f) return 5; // Reduce memory
usage
            return 0; // No optimization needed
        }
    }

    [BurstCompile]
    struct MaldivianSceneOptimization : IJob
```

```
{
    public NativeArray<float3> palmTreePositions;
    public NativeArray<float3> buildingPositions;
    public NativeArray<float3> boatPositions;
    public NativeArray<float3> oceanPositions;
    public NativeArray<float3> terrainPositions;

    public NativeArray<int> lodRecommendations;
    public NativeArray<bool> cullingRecommendations;
    public NativeArray<int> batchingRecommendations;

    public NativeArray<float> distanceFromCamera;
    public NativeArray<bool> isVisible;
    public NativeArray<int> optimizationPriority;

    public float3 cameraPosition;
    public float maxDrawDistance;
    public float minDetailDistance;
    public int targetFrameRate;

    public void Execute()
    {
        CalculateLODLevels();
        DetermineCullingNeeds();
        CalculateBatchingOpportunities();
        CalculateOptimizationPriorities();
    }

    [BurstCompile]
    void CalculateLODLevels()
    {
```

```
        // Calculate appropriate LOD levels for Maldivian scene elements
        for (int i = 0; i < palmTreePositions.Length; i++)
        {
            float distance = math.length(palmTreePositions[i] - cameraPosition);
            int lod = CalculateLOD(distance);
            lodRecommendations[i] = lod;
            distanceFromCamera[i] = distance;
        }

        for (int i = 0; i < buildingPositions.Length; i++)
        {
            float distance = math.length(buildingPositions[i] - cameraPosition);
            int lod = CalculateLOD(distance);
            lodRecommendations[palmTreePositions.Length + i] = lod;
            distanceFromCamera[palmTreePositions.Length + i] = distance;
        }

        for (int i = 0; i < boatPositions.Length; i++)
        {
            float distance = math.length(boatPositions[i] - cameraPosition);
            int lod = CalculateLOD(distance);
            lodRecommendations[palmTreePositions.Length +
buildingPositions.Length + i] = lod;
            distanceFromCamera[palmTreePositions.Length +
buildingPositions.Length + i] = distance;
        }
    }

    [BurstCompile]
    void DetermineCullingNeeds()
    {
```

```
        // Determine which objects should be culled
        int totalObjects = palmTreePositions.Length + buildingPositions.Length +
boatPositions.Length + oceanPositions.Length + terrainPositions.Length;

        for (int i = 0; i < totalObjects; i++)
        {
            float distance = i < distanceFromCamera.Length ?
distanceFromCamera[i] : maxDrawDistance + 1;
            bool shouldCull = distance > maxDrawDistance;
            cullingRecommendations[i] = shouldCull;
            isVisible[i] = !shouldCull;
        }
    }

    [BurstCompile]
    void CalculateBatchingOpportunities()
    {
        // Calculate static/dynamic batching opportunities
        int batchIndex = 0;

        // Palm trees can be batched
        for (int i = 0; i < palmTreePositions.Length; i++)
        {
            if (distanceFromCamera[i] < maxDrawDistance)
            {
                batchingRecommendations[batchIndex++] = 1; // Static batching
            }
        }

        // Buildings can be batched
        for (int i = 0; i < buildingPositions.Length; i++)
```

```
            {
                int index = palmTreePositions.Length + i;
                if (distanceFromCamera[index] < maxDrawDistance)
                {
                    batchingRecommendations[batchIndex++] = 1; // Static batching
                }
            }
        }

        [BurstCompile]
        void CalculateOptimizationPriorities()
        {
            // Calculate optimization priorities based on performance impact
            for (int i = 0; i < optimizationPriority.Length; i++)
            {
                float distance = i < distanceFromCamera.Length ?
distanceFromCamera[i] : maxDrawDistance;
                bool visible = i < isVisible.Length ? isVisible[i] : false;

                int priority = 0;
                if (visible && distance < minDetailDistance) priority = 3; // High priority
                else if (visible && distance < maxDrawDistance * 0.5f) priority = 2; //
Medium priority
                else if (visible) priority = 1; // Low priority
                else priority = 0; // Not visible

                optimizationPriority[i] = priority;
            }
        }

        [BurstCompile]
```

```
    int CalculateLOD(float distance)
    {
        if (distance < minDetailDistance) return 0; // High detail
        if (distance < maxDrawDistance * 0.5f) return 1; // Medium detail
        if (distance < maxDrawDistance) return 2; // Low detail
        return 3; // Culled
    }
}

public static PerformanceProfiler Instance { get; private set; }

[Header("Performance Settings")]
public bool enableProfiling = true;
public bool autoOptimize = true;
public bool enableMobileOptimizations = true;
public MobileTier targetMobileTier = MobileTier.MidRange;
public int targetFrameRate = 60;

[Header("Optimization Targets")]
public float maxFrameTime = 0.0167f; // 60 FPS
public int maxDrawCalls = 100;
public int maxTriangleCount = 50000;
public float maxMemoryUsage = 200f; // MB
public int maxTextureMemory = 100f; // MB

[Header("Maldivian Scene Optimization")]
public bool optimizePalmTrees = true;
public bool optimizeBuildings = true;
public bool optimizeOcean = true;
public bool optimizeTerrain = true;
public bool respectCulturalAccuracy = true;
```

```
[Header("LOD Settings")]
public float maxDrawDistance = 500f;
public float minDetailDistance = 50f;
public bool enableDynamicLOD = true;
public float lodUpdateInterval = 2f;

private ProfilerRecorder cpuTimeRecorder;
private ProfilerRecorder gpuTimeRecorder;
private ProfilerRecorder drawCallRecorder;
private ProfilerRecorder triangleRecorder;
private ProfilerRecorder memoryRecorder;

private float[] cpuTimeHistory;
private float[] gpuTimeHistory;
private int[] drawCallHistory;
private int[] triangleCountHistory;
private float[] memoryUsageHistory;
private int historyIndex = 0;

public enum MobileTier
{
    LowEnd,
    MidRange,
    HighEnd
}

public enum OptimizationRecommendation
{
    None = 0,
    ReduceCPULoad = 1,
```

```
        ReduceGPULoad = 2,
        ReduceDrawCalls = 3,
        ReduceTriangles = 4,
        ReduceMemory = 5,
        EnableLOD = 6,
        EnableCulling = 7,
        EnableBatching = 8
    }

    void Awake()
    {
        Instance = this;
        InitializePerformanceProfiler();
    }

    void InitializePerformanceProfiler()
    {
        // Initialize history arrays
        int historySize = 60;
        cpuTimeHistory = new float[historySize];
        gpuTimeHistory = new float[historySize];
        drawCallHistory = new int[historySize];
        triangleCountHistory = new int[historySize];
        memoryUsageHistory = new float[historySize];

        // Initialize Unity Profiler recorders
        cpuTimeRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Internal,
"CPU Time");
        gpuTimeRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Internal,
"GPU Time");
```

```
        drawCallRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Render,
"Draw Calls Count");
        triangleRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Render,
"Triangles Count");
        memoryRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Memory,
"Total Used Memory");

        // Initialize performance analysis
        int sampleCount = 60;
        var cpuTimes = new NativeArray<float>(sampleCount, Allocator.TempJob);
        var gpuTimes = new NativeArray<float>(sampleCount, Allocator.TempJob);
        var drawCalls = new NativeArray<int>(sampleCount, Allocator.TempJob);
        var triangles = new NativeArray<int>(sampleCount, Allocator.TempJob);
        var memoryUsages = new NativeArray<float>(sampleCount,
Allocator.TempJob);
        var scores = new NativeArray<float>(sampleCount, Allocator.TempJob);
        var bottlenecks = new NativeArray<bool>(sampleCount,
Allocator.TempJob);
        var recommendations = new NativeArray<int>(sampleCount,
Allocator.TempJob);

        // Initialize with sample data
        for (int i = 0; i < sampleCount; i++)
        {
            cpuTimes[i] = UnityEngine.Random.Range(0.008f, 0.016f);
            gpuTimes[i] = UnityEngine.Random.Range(0.005f, 0.012f);
            drawCalls[i] = UnityEngine.Random.Range(30, 80);
            triangles[i] = UnityEngine.Random.Range(10000, 40000);
            memoryUsages[i] = UnityEngine.Random.Range(80f, 150f);
        }
```

```
        var optimizationJob = new MobilePerformanceOptimization
        {
            cpuTimeHistory = cpuTimes,
            gpuTimeHistory = gpuTimes,
            drawCallHistory = drawCalls,
            triangleCountHistory = triangles,
            memoryUsageHistory = memoryUsages,
            optimizationScores = scores,
            performanceBottlenecks = bottlenecks,
            optimizationRecommendations = recommendations,
            targetFrameTime = maxFrameTime,
            maxDrawCalls = maxDrawCalls,
            maxTriangles = maxTriangleCount,
            maxMemoryUsage = maxMemoryUsage,
            mobileTier = targetMobileTier
        };

        // Initialize Maldivian scene optimization
        int palmTrees = 50;
        int buildings = 30;
        int boats = 20;
        int oceanTiles = 100;
        int terrainChunks = 25;

        var palmPositions = new NativeArray<float3>(palmTrees,
Allocator.TempJob);
        var buildingPositions = new NativeArray<float3>(buildings,
Allocator.TempJob);
        var boatPositions = new NativeArray<float3>(boats, Allocator.TempJob);
        var oceanPositions = new NativeArray<float3>(oceanTiles,
Allocator.TempJob);
```

```
        var terrainPositions = new NativeArray<float3>(terrainChunks,
Allocator.TempJob);
        var lodRecommendations = new NativeArray<int>(palmTrees + buildings +
boats + oceanTiles + terrainChunks, Allocator.TempJob);
        var cullingRecommendations = new NativeArray<bool>(palmTrees +
buildings + boats + oceanTiles + terrainChunks, Allocator.TempJob);
        var batchingRecommendations = new NativeArray<int>(palmTrees +
buildings + boats + oceanTiles + terrainChunks, Allocator.TempJob);
        var distances = new NativeArray<float>(palmTrees + buildings + boats,
Allocator.TempJob);
        var visibility = new NativeArray<bool>(palmTrees + buildings + boats +
oceanTiles + terrainChunks, Allocator.TempJob);
        var priorities = new NativeArray<int>(palmTrees + buildings + boats +
oceanTiles + terrainChunks, Allocator.TempJob);

        // Initialize with sample scene data
        for (int i = 0; i < palmTrees; i++)
        {
            palmPositions[i] = UnityEngine.Random.insideUnitSphere * 200f;
        }
        for (int i = 0; i < buildings; i++)
        {
            buildingPositions[i] = UnityEngine.Random.insideUnitSphere * 150f;
        }
        for (int i = 0; i < boats; i++)
        {
            boatPositions[i] = UnityEngine.Random.insideUnitSphere * 100f;
        }
        for (int i = 0; i < oceanTiles; i++)
        {
            oceanPositions[i] = UnityEngine.Random.insideUnitSphere * 300f;
```

```
        }
        for (int i = 0; i < terrainChunks; i++)
        {
            terrainPositions[i] = UnityEngine.Random.insideUnitSphere * 250f;
        }

        var sceneJob = new MaldivianSceneOptimization
        {
            palmTreePositions = palmPositions,
            buildingPositions = buildingPositions,
            boatPositions = boatPositions,
            oceanPositions = oceanPositions,
            terrainPositions = terrainPositions,
            lodRecommendations = lodRecommendations,
            cullingRecommendations = cullingRecommendations,
            batchingRecommendations = batchingRecommendations,
            distanceFromCamera = distances,
            isVisible = visibility,
            optimizationPriority = priorities,
            cameraPosition = Camera.main ? Camera.main.transform.position :
Vector3.zero,
            maxDrawDistance = maxDrawDistance,
            minDetailDistance = minDetailDistance,
            targetFrameRate = targetFrameRate
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(sampleCount,
8);
        JobHandle sceneHandle = sceneJob.Schedule(optimizationHandle);
        sceneHandle.Complete();
```

```
        // Process results
        ProcessOptimizationResults(scores, bottlenecks, recommendations);
        ProcessSceneOptimization(lodRecommendations,
cullingRecommendations, batchingRecommendations, distances, visibility);

        // Cleanup
        cpuTimes.Dispose();
        gpuTimes.Dispose();
        drawCalls.Dispose();
        triangles.Dispose();
        memoryUsages.Dispose();
        scores.Dispose();
        bottlenecks.Dispose();
        recommendations.Dispose();
        palmPositions.Dispose();
        buildingPositions.Dispose();
        boatPositions.Dispose();
        oceanPositions.Dispose();
        terrainPositions.Dispose();
        lodRecommendations.Dispose();
        cullingRecommendations.Dispose();
        batchingRecommendations.Dispose();
        distances.Dispose();
        visibility.Dispose();
        priorities.Dispose();

        StartPerformanceMonitoring();
    }

    void ProcessOptimizationResults(NativeArray<float> scores,
NativeArray<bool> bottlenecks, NativeArray<int> recommendations)
```

```
    {
        // Process optimization analysis results
        float avgScore = 0f;
        int bottleneckCount = 0;
        Dictionary<int, int> recommendationCounts = new Dictionary<int, int>();

        for (int i = 0; i < scores.Length; i++)
        {
            avgScore += scores[i];
            if (bottlenecks[i]) bottleneckCount++;

            int rec = recommendations[i];
            if (recommendationCounts.ContainsKey(rec))
                recommendationCounts[rec]++;
            else
                recommendationCounts[rec] = 1;
        }

        avgScore /= scores.Length;

        Debug.Log($"Performance optimization: Average score {avgScore:F2},
{bottleneckCount} bottlenecks detected");
    }

    void ProcessSceneOptimization(NativeArray<int> lod, NativeArray<bool>
culling, NativeArray<int> batching, NativeArray<float> distances,
NativeArray<bool> visibility)
    {
        // Process scene optimization results
        int totalOptimized = 0;
        int culledObjects = 0;
```

```
        for (int i = 0; i < lod.Length; i++)
        {
            if (lod[i] > 0) totalOptimized++;
        }

        for (int i = 0; i < culling.Length; i++)
        {
            if (culling[i]) culledObjects++;
        }

        Debug.Log($"Scene optimization: {totalOptimized} objects optimized,
{culledObjects} objects culled");
    }

    void StartPerformanceMonitoring()
    {
        // Start performance monitoring
        InvokeRepeating("UpdatePerformanceMetrics", 0.0f, 1.0f / 60.0f); // Every
frame
        InvokeRepeating("AnalyzePerformance", 1.0f, 5.0f);
        InvokeRepeating("ApplyOptimizations", 5.0f, 10.0f);
    }

    void UpdatePerformanceMetrics()
    {
        if (!enableProfiling) return;

        // Record current performance metrics
        cpuTimeHistory[historyIndex] = cpuTimeRecorder.LastValue / 1000000f; //
Convert nanoseconds to milliseconds
```

```
        gpuTimeHistory[historyIndex] = gpuTimeRecorder.LastValue / 1000000f;
        drawCallHistory[historyIndex] = drawCallRecorder.LastValue;
        triangleCountHistory[historyIndex] = triangleRecorder.LastValue;
        memoryUsageHistory[historyIndex] = memoryRecorder.LastValue / (1024f *
1024f); // Convert to MB

        historyIndex = (historyIndex + 1) % cpuTimeHistory.Length;
    }

    void AnalyzePerformance()
    {
        if (!enableProfiling) return;

        // Analyze recent performance data
        float avgCPU = 0f, avgGPU = 0f, avgMemory = 0f;
        int avgDrawCalls = 0, avgTriangles = 0;

        for (int i = 0; i < cpuTimeHistory.Length; i++)
        {
            avgCPU += cpuTimeHistory[i];
            avgGPU += gpuTimeHistory[i];
            avgDrawCalls += drawCallHistory[i];
            avgTriangles += triangleCountHistory[i];
            avgMemory += memoryUsageHistory[i];
        }

        int sampleCount = cpuTimeHistory.Length;
        avgCPU /= sampleCount;
        avgGPU /= sampleCount;
        avgDrawCalls /= sampleCount;
        avgTriangles /= sampleCount;
```

```
        avgMemory /= sampleCount;

        // Check for performance issues
        bool cpuIssue = avgCPU > maxFrameTime * 1000f * 0.7f;
        bool gpuIssue = avgGPU > maxFrameTime * 1000f * 0.7f;
        bool drawCallIssue = avgDrawCalls > maxDrawCalls * 0.8f;
        bool triangleIssue = avgTriangles > maxTriangleCount * 0.8f;
        bool memoryIssue = avgMemory > maxMemoryUsage * 0.8f;

        if (cpuIssue || gpuIssue || drawCallIssue || triangleIssue || memoryIssue)
        {
            Debug.LogWarning($"Performance issues detected:
CPU={avgCPU:F1}ms, GPU={avgGPU:F1}ms, DrawCalls={avgDrawCalls},
Triangles={avgTriangles}, Memory={avgMemory:F1}MB");
        }
    }

    void ApplyOptimizations()
    {
        if (!autoOptimize) return;

        // Apply automatic optimizations based on analysis
        ApplyLODOptimizations();
        ApplyCullingOptimizations();
        ApplyBatchingOptimizations();
    }

    void ApplyLODOptimizations()
    {
        if (!enableDynamicLOD) return;
```

```
        // Apply LOD recommendations to scene objects
        // Implementation would adjust LOD levels based on distance and
performance
    }

    void ApplyCullingOptimizations()
    {
        // Apply culling recommendations
        // Implementation would enable/disable renderers based on visibility
    }

    void ApplyBatchingOptimizations()
    {
        // Apply static/dynamic batching
        // Implementation would group similar objects for batching
    }

    public void ForceOptimization()
    {
        AnalyzePerformance();
        ApplyOptimizations();
    }

    public PerformanceSnapshot GetPerformanceSnapshot()
    {
        float currentCPU = cpuTimeRecorder.LastValue / 1000000f;
        float currentGPU = gpuTimeRecorder.LastValue / 1000000f;
        int currentDrawCalls = drawCallRecorder.LastValue;
        int currentTriangles = triangleRecorder.LastValue;
        float currentMemory = memoryRecorder.LastValue / (1024f * 1024f);
```

```
        return new PerformanceSnapshot
        {
            profiling_enabled = enableProfiling,
            auto_optimization = autoOptimize,
            current_fps = 1.0f / Time.unscaledDeltaTime,
            cpu_time_ms = currentCPU,
            gpu_time_ms = currentGPU,
            draw_calls = currentDrawCalls,
            triangle_count = currentTriangles,
            memory_usage_mb = currentMemory,
            optimization_score = CalculateOptimizationScore()
        };
    }

    float CalculateOptimizationScore()
    {
        float cpuScore = Mathf.Clamp01(1.0f - (cpuTimeRecorder.LastValue /
1000000f) / (maxFrameTime * 1000f));
        float gpuScore = Mathf.Clamp01(1.0f - (gpuTimeRecorder.LastValue /
1000000f) / (maxFrameTime * 1000f));
        float drawCallScore = Mathf.Clamp01(1.0f -
(float)drawCallRecorder.LastValue / (float)maxDrawCalls);
        float memoryScore = Mathf.Clamp01(1.0f - (memoryRecorder.LastValue /
(1024f * 1024f)) / maxMemoryUsage);

        return (cpuScore + gpuScore + drawCallScore + memoryScore) * 0.25f;
    }

    [System.Serializable]
    public class PerformanceSnapshot
    {
```

```
        public bool profiling_enabled;
        public bool auto_optimization;
        public float current_fps;
        public float cpu_time_ms;
        public float gpu_time_ms;
        public int draw_calls;
        public int triangle_count;
        public float memory_usage_mb;
        public float optimization_score;
    }
}
```

44. AssetBundleManager.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;
using System.IO;
using System;

[BurstCompile]
public class AssetBundleManager : MonoBehaviour
{
    [BurstCompile]
    struct ContentStreamingOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> assetSizes;
```

```
[ReadOnly] public NativeArray<int> assetPriorities;
[ReadOnly] public NativeArray<bool> isEssential;
[ReadOnly] public NativeArray<float> lastAccessTimes;
[WriteOnly] public NativeArray<bool> shouldStream;
[WriteOnly] public NativeArray<int> streamingOrder;
[WriteOnly] public NativeArray<float> loadingScores;

public float currentTime;
public float availableBandwidth;
public float memoryBudget;
public float priorityWeight;

public void Execute(int index)
{
    float size = assetSizes[index];
    int priority = assetPriorities[index];
    bool essential = isEssential[index];
    float lastAccess = lastAccessTimes[index];

    // Calculate streaming score
    float score = CalculateLoadingScore(size, priority, essential, lastAccess);
    bool shouldLoad = ShouldStreamAsset(score, size);
    int order = CalculateStreamingOrder(priority, essential, lastAccess);

    loadingScores[index] = score;
    shouldStream[index] = shouldLoad;
    streamingOrder[index] = order;
}

[BurstCompile]
```

```
    float CalculateLoadingScore(float size, int priority, bool essential, float
lastAccess)
    {
        float baseScore = 0.5f;

        // Priority scoring
        baseScore += (priority / 10.0f) * 0.3f;

        // Essential assets get higher scores
        if (essential) baseScore += 0.2f;

        // Recency scoring (more recently accessed assets)
        float timeSinceAccess = currentTime - lastAccess;
        float recencyScore = math.saturate(1.0f - (timeSinceAccess / 3600.0f)); //
1 hour decay
        baseScore += recencyScore * 0.1f;

        // Size penalty (larger assets get slightly lower scores)
        float sizePenalty = math.saturate(size / 50.0f) * 0.2f; // 50MB baseline
        baseScore = math.max(0.0f, baseScore - sizePenalty);

        return math.saturate(baseScore);
    }

    [BurstCompile]
    bool ShouldStreamAsset(float score, float size)
    {
        // Check if asset should be streamed based on score and constraints
        bool scoreOk = score > 0.6f;
        bool bandwidthOk = size < availableBandwidth * 0.1f; // 10% of bandwidth
per asset
```

```
            bool memoryOk = size < memoryBudget * 0.05f; // 5% of memory per
asset

            return scoreOk && bandwidthOk && memoryOk;
        }

        [BurstCompile]
        int CalculateStreamingOrder(int priority, bool essential, float lastAccess)
        {
            // Calculate streaming order (lower number = higher priority)
            int order = priority * 100;
            if (essential) order -= 1000;
            order -= Mathf.RoundToInt((1.0f - (currentTime - lastAccess) / 3600.0f) *
10);

            return math.max(0, order);
        }
    }

    [BurstCompile]
    struct MaldivianAssetOptimization : IJob
    {
        public NativeArray<int> islandAssetBundles;
        public NativeArray<int> culturalAssetBundles;
        public NativeArray<int> traditionalAssetBundles;
        public NativeArray<int> religiousAssetBundles;
        public NativeArray<int> environmentalAssetBundles;

        public NativeArray<bool> culturalPriorityFlags;
        public NativeArray<float> authenticityScores;
        public NativeArray<int> respectfulLoadingOrder;
```

```
public int currentIslandID;
public bool isPrayerTime;
public bool isCulturalEvent;
public float culturalSensitivityWeight;

public void Execute()
{
    PrioritizeIslandAssets();
    PrioritizeCulturalAssets();
    CalculateRespectfulLoadingOrder();
    ValidateCulturalAuthenticity();
}

[BurstCompile]
void PrioritizeIslandAssets()
{
    // Prioritize assets for current island
    for (int i = 0; i < islandAssetBundles.Length; i++)
    {
        bool isCurrentIsland = i == currentIslandID;
        bool isNearbyIsland = math.abs(i - currentIslandID) <= 3;

        if (isCurrentIsland)
        {
            culturalPriorityFlags[i] = true;
            authenticityScores[i] = 0.95f;
        }
        else if (isNearbyIsland)
        {
            culturalPriorityFlags[i] = true;
```

```
            authenticityScores[i] = 0.80f;
        }
        else
        {
            culturalPriorityFlags[i] = false;
            authenticityScores[i] = 0.60f;
        }
    }
}

[BurstCompile]
void PrioritizeCulturalAssets()
{
    // Prioritize cultural and religious assets
    int offset = islandAssetBundles.Length;

    for (int i = 0; i < culturalAssetBundles.Length; i++)
    {
        bool highPriority = isPrayerTime || isCulturalEvent;
        culturalPriorityFlags[offset + i] = highPriority;
        authenticityScores[offset + i] = highPriority ? 0.95f : 0.75f;
    }

    // Religious assets get highest priority during prayer times
    offset += culturalAssetBundles.Length;
    for (int i = 0; i < religiousAssetBundles.Length; i++)
    {
        bool highestPriority = isPrayerTime;
        culturalPriorityFlags[offset + i] = highestPriority;
        authenticityScores[offset + i] = highestPriority ? 1.0f : 0.90f;
    }
```

```
    }

    [BurstCompile]
    void CalculateRespectfulLoadingOrder()
    {
        // Calculate loading order that respects cultural priorities
        int index = 0;

        // Religious assets first during prayer times
        if (isPrayerTime)
        {
            for (int i = 0; i < religiousAssetBundles.Length; i++)
            {
                respectfulLoadingOrder[index++] = islandAssetBundles.Length +
culturalAssetBundles.Length + i;
            }
        }

        // Current island assets
        for (int i = 0; i < islandAssetBundles.Length; i++)
        {
            if (i == currentIslandID)
            {
                respectfulLoadingOrder[index++] = i;
            }
        }

        // Nearby islands
        for (int i = 0; i < islandAssetBundles.Length; i++)
        {
            if (math.abs(i - currentIslandID) <= 3 && i != currentIslandID)
```

```
            {
                respectfulLoadingOrder[index++] = i;
            }
        }

        // Cultural assets
        int culturalOffset = islandAssetBundles.Length;
        for (int i = 0; i < culturalAssetBundles.Length; i++)
        {
            respectfulLoadingOrder[index++] = culturalOffset + i;
        }

        // Environmental assets
        int envOffset = islandAssetBundles.Length + culturalAssetBundles.Length
+ religiousAssetBundles.Length;
        for (int i = 0; i < environmentalAssetBundles.Length; i++)
        {
            respectfulLoadingOrder[index++] = envOffset + i;
        }
    }

    [BurstCompile]
    void ValidateCulturalAuthenticity()
    {
        // Validate cultural authenticity of assets
        int totalCulturalAssets = culturalAssetBundles.Length +
traditionalAssetBundles.Length + religiousAssetBundles.Length;

        for (int i = 0; i < totalCulturalAssets; i++)
        {
            bool isAuthentic = ValidateAssetAuthenticity(i);
```

```
            if (isAuthentic)
            {
                authenticityScores[islandAssetBundles.Length + i] *= 1.1f; // Boost
authentic assets
                authenticityScores[islandAssetBundles.Length + i] = math.min(1.0f,
authenticityScores[islandAssetBundles.Length + i]);
            }
        }
    }

    [BurstCompile]
    bool ValidateAssetAuthenticity(int assetIndex)
    {
        // Validate cultural authenticity of specific assets
        // Simplified validation - would check against cultural database
        return assetIndex >= 0 && assetIndex < 1000; // Valid asset range
    }
}

public static AssetBundleManager Instance { get; private set; }

[Header("Asset Bundle Settings")]
public bool enableStreaming = true;
public bool enableCaching = true;
public int maxCacheSize = 500; // MB
public float streamingSpeed = 1.0f;
public bool enableCompression = true;

[Header("Cultural Content Priority")]
public bool prioritizeCulturalAssets = true;
public bool respectPrayerTimeLoading = true;
```

```
public bool maintainCulturalAuthenticity = true;
public bool enableRespectfulStreaming = true;

[Header("Mobile Optimization")]
public bool enableMobileStreaming = true;
public int mobileMaxConcurrent = 2;
public float mobileBandwidthLimit = 2.0f; // MB/s
public bool pauseOnMobileNetwork = true;

private Dictionary<string, AssetBundle> loadedBundles;
private Dictionary<string, float> bundleSizes;
private Dictionary<string, DateTime> accessTimes;
private Queue<string> loadQueue;
private List<string> loadingBundles;
private float currentCacheUsage;

public enum BundleType
{
    IslandContent,
    CulturalContent,
    ReligiousContent,
    EnvironmentalContent,
    TraditionalContent,
    EssentialContent,
    OptionalContent
}

void Awake()
{
    Instance = this;
    InitializeAssetBundleManager();
```

```
    }

    void InitializeAssetBundleManager()
    {
        // Initialize collections
        loadedBundles = new Dictionary<string, AssetBundle>();
        bundleSizes = new Dictionary<string, float>();
        accessTimes = new Dictionary<string, DateTime>();
        loadQueue = new Queue<string>();
        loadingBundles = new List<string>();

        // Initialize content streaming optimization
        int assetCount = 100;
        var sizes = new NativeArray<float>(assetCount, Allocator.TempJob);
        var priorities = new NativeArray<int>(assetCount, Allocator.TempJob);
        var essential = new NativeArray<bool>(assetCount, Allocator.TempJob);
        var lastAccess = new NativeArray<float>(assetCount, Allocator.TempJob);
        var shouldStream = new NativeArray<bool>(assetCount,
Allocator.TempJob);
        var streamOrder = new NativeArray<int>(assetCount, Allocator.TempJob);
        var scores = new NativeArray<float>(assetCount, Allocator.TempJob);

        // Initialize with sample asset data
        for (int i = 0; i < assetCount; i++)
        {
            sizes[i] = UnityEngine.Random.Range(1f, 20f); // 1-20 MB
            priorities[i] = UnityEngine.Random.Range(1, 10);
            essential[i] = i < 20; // First 20 are essential
            lastAccess[i] = Time.time - UnityEngine.Random.Range(0f, 3600f); // Last
hour
        }
```

```
        var streamingJob = new ContentStreamingOptimization
        {
            assetSizes = sizes,
            assetPriorities = priorities,
            isEssential = essential,
            lastAccessTimes = lastAccess,
            shouldStream = shouldStream,
            streamingOrder = streamOrder,
            loadingScores = scores,
            currentTime = Time.time,
            availableBandwidth = mobileBandwidthLimit * 1024f * 1024f, // Convert to
bytes
            memoryBudget = maxCacheSize * 1024f * 1024f,
            priorityWeight = prioritizeCulturalAssets ? 2.0f : 1.0f
        };

        // Initialize Maldivian asset optimization
        int islands = 41;
        int culturalBundles = 15;
        int traditionalBundles = 10;
        int religiousBundles = 8;
        int environmentalBundles = 20;

        var islandBundles = new NativeArray<int>(islands, Allocator.TempJob);
        var culturalBundlesArray = new NativeArray<int>(culturalBundles,
Allocator.TempJob);
        var traditionalBundlesArray = new NativeArray<int>(traditionalBundles,
Allocator.TempJob);
        var religiousBundlesArray = new NativeArray<int>(religiousBundles,
Allocator.TempJob);
```

```
        var environmentalBundlesArray = new
NativeArray<int>(environmentalBundles, Allocator.TempJob);
        var culturalFlags = new NativeArray<bool>(islands + culturalBundles +
traditionalBundles + religiousBundles + environmentalBundles,
Allocator.TempJob);
        var authenticityScores = new NativeArray<float>(islands + culturalBundles +
traditionalBundles + religiousBundles + environmentalBundles,
Allocator.TempJob);
        var respectfulOrder = new NativeArray<int>(islands + culturalBundles +
traditionalBundles + religiousBundles + environmentalBundles,
Allocator.TempJob);

        // Initialize with bundle IDs
        for (int i = 0; i < islands; i++) islandBundles[i] = i;
        for (int i = 0; i < culturalBundles; i++) culturalBundlesArray[i] = i + 100;
        for (int i = 0; i < traditionalBundles; i++) traditionalBundlesArray[i] = i + 200;
        for (int i = 0; i < religiousBundles; i++) religiousBundlesArray[i] = i + 300;
        for (int i = 0; i < environmentalBundles; i++) environmentalBundlesArray[i] =
i + 400;

        var culturalJob = new MaldivianAssetOptimization
        {
            islandAssetBundles = islandBundles,
            culturalAssetBundles = culturalBundlesArray,
            traditionalAssetBundles = traditionalBundlesArray,
            religiousAssetBundles = religiousBundlesArray,
            environmentalAssetBundles = environmentalBundlesArray,
            culturalPriorityFlags = culturalFlags,
            authenticityScores = authenticityScores,
            respectfulLoadingOrder = respectfulOrder,
            currentIslandID = 0,
```

```
            isPrayerTime = IsPrayerTime(),
            isCulturalEvent = IsCulturalEvent(),
            culturalSensitivityWeight = maintainCulturalAuthenticity ? 2.0f : 1.0f
        };

        JobHandle streamingHandle = streamingJob.Schedule(assetCount, 8);
        JobHandle culturalHandle = culturalJob.Schedule(streamingHandle);
        culturalHandle.Complete();

        // Process results
        ProcessStreamingResults(shouldStream, streamOrder, scores);
        ProcessCulturalOptimization(culturalFlags, authenticityScores,
respectfulOrder);

        // Cleanup
        sizes.Dispose();
        priorities.Dispose();
        essential.Dispose();
        lastAccess.Dispose();
        shouldStream.Dispose();
        streamOrder.Dispose();
        scores.Dispose();
        islandBundles.Dispose();
        culturalBundlesArray.Dispose();
        traditionalBundlesArray.Dispose();
        religiousBundlesArray.Dispose();
        environmentalBundlesArray.Dispose();
        culturalFlags.Dispose();
        authenticityScores.Dispose();
        respectfulOrder.Dispose();
```

```
        StartAssetStreaming();
    }

    void ProcessStreamingResults(NativeArray<bool> shouldStream,
NativeArray<int> streamOrder, NativeArray<float> scores)
    {
        // Process streaming optimization results
        int streamCount = 0;
        for (int i = 0; i < shouldStream.Length; i++)
        {
            if (shouldStream[i]) streamCount++;
        }
        Debug.Log($"Content streaming: {streamCount} assets queued for
loading");
    }

    void ProcessCulturalOptimization(NativeArray<bool> culturalFlags,
NativeArray<float> authenticity, NativeArray<int> order)
    {
        // Process cultural optimization results
        int culturalCount = 0;
        for (int i = 0; i < culturalFlags.Length; i++)
        {
            if (culturalFlags[i]) culturalCount++;
        }
        Debug.Log($"Cultural optimization: {culturalCount} assets flagged for
cultural priority");
    }

    void StartAssetStreaming()
    {
```

```
        // Start asset streaming monitoring
        InvokeRepeating("UpdateStreamingQueue", 1.0f, 2.0f);
        InvokeRepeating("ManageCache", 30.0f, 60.0f);
        InvokeRepeating("ValidateCulturalLoading", 5.0f, 15.0f);
    }

    public void LoadAssetBundle(string bundleName, BundleType bundleType,
Action<AssetBundle> onComplete)
    {
        if (!enableStreaming)
        {
            onComplete?.Invoke(null);
            return;
        }

        // Check if already loaded
        if (loadedBundles.ContainsKey(bundleName))
        {
            UpdateAccessTime(bundleName);
            onComplete?.Invoke(loadedBundles[bundleName]);
            return;
        }

        // Check if currently loading
        if (loadingBundles.Contains(bundleName))
        {
            // Add callback to existing load operation
            return;
        }

        // Queue for loading
```

```
        QueueBundleForLoading(bundleName, bundleType, onComplete);
    }

    void QueueBundleForLoading(string bundleName, BundleType bundleType,
Action<AssetBundle> onComplete)
    {
        // Determine loading priority based on cultural significance
        int priority = CalculateCulturalPriority(bundleName, bundleType);

        // Add to loading queue
        loadQueue.Enqueue(bundleName);
        loadingBundles.Add(bundleName);

        // Store callback for completion
        StartCoroutine(LoadBundleAsync(bundleName, bundleType, onComplete));
    }

    int CalculateCulturalPriority(string bundleName, BundleType bundleType)
    {
        int basePriority = (int)bundleType;

        // Boost cultural content priority
        if (prioritizeCulturalAssets)
        {
            switch (bundleType)
            {
                case BundleType.ReligiousContent:
                    basePriority += IsPrayerTime() ? 100 : 50;
                    break;
                case BundleType.CulturalContent:
                case BundleType.TraditionalContent:
```

```
                basePriority += 30;
                break;
            }
        }

        return basePriority;
    }

    System.Collections.IEnumerator LoadBundleAsync(string bundleName,
BundleType bundleType, Action<AssetBundle> onComplete)
    {
        string path = GetBundlePath(bundleName);

        // Check bandwidth constraints on mobile
        if (enableMobileStreaming && Application.isMobilePlatform)
        {
            while (GetActiveDownloads() >= mobileMaxConcurrent)
            {
                yield return new WaitForSeconds(0.5f);
            }
        }

        // Load bundle
        AssetBundleCreateRequest request =
AssetBundle.LoadFromFileAsync(path);

        while (!request.isDone)
        {
            yield return null;

            // Check for cancellation (prayer time, etc.)
```

```
            if (respectPrayerTimeLoading && IsPrayerTime())
            {
                if (bundleType != BundleType.ReligiousContent && bundleType !=
BundleType.EssentialContent)
                {
                    // Pause non-essential loading during prayer times
                    yield return new WaitWhile(() => IsPrayerTime());
                }
            }
        }

        if (request.assetBundle != null)
        {
            loadedBundles[bundleName] = request.assetBundle;
            accessTimes[bundleName] = DateTime.Now;

            // Estimate bundle size
            bundleSizes[bundleName] = EstimateBundleSize(request.assetBundle);
            currentCacheUsage += bundleSizes[bundleName];

            onComplete?.Invoke(request.assetBundle);
        }
        else
        {
            Debug.LogError($"Failed to load asset bundle: {bundleName}");
            onComplete?.Invoke(null);
        }

        loadingBundles.Remove(bundleName);
    }
```

```
    string GetBundlePath(string bundleName)
    {
        // Determine platform-specific bundle path
        string platform = GetPlatformName();
        return Path.Combine(Application.streamingAssetsPath, platform,
bundleName);
    }

    string GetPlatformName()
    {
        return Application.platform switch
        {
            RuntimePlatform.Android => "Android",
            RuntimePlatform.IPhonePlayer => "iOS",
            RuntimePlatform.WindowsPlayer => "Windows",
            RuntimePlatform.OSXPlayer => "MacOS",
            _ => "Standalone"
        };
    }

    float EstimateBundleSize(AssetBundle bundle)
    {
        // Estimate bundle size based on loaded assets
        // Simplified estimation
        return 10.0f; // 10 MB default
    }

    int GetActiveDownloads()
    {
        return loadingBundles.Count;
    }
```

```
    void UpdateStreamingQueue()
    {
        // Process loading queue
        while (loadQueue.Count > 0 && GetActiveDownloads() <
mobileMaxConcurrent)
        {
            string bundleName = loadQueue.Dequeue();
            // Continue loading process
        }
    }

    void ManageCache()
    {
        // Manage cache size and remove old bundles
        if (currentCacheUsage > maxCacheSize)
        {
            RemoveLeastUsedBundles();
        }
    }

    void RemoveLeastUsedBundles()
    {
        // Remove least recently used bundles to free cache space
        List<string> bundlesToRemove = new List<string>();

        foreach (var kvp in accessTimes)
        {
            if (DateTime.Now - kvp.Value > TimeSpan.FromMinutes(30)) // 30
minutes unused
            {
```

```
            bundlesToRemove.Add(kvp.Key);
        }
    }

    foreach (string bundleName in bundlesToRemove)
    {
        UnloadBundle(bundleName);
    }
}

public void UnloadBundle(string bundleName)
{
    if (loadedBundles.ContainsKey(bundleName))
    {
        loadedBundles[bundleName].Unload(true);
        loadedBundles.Remove(bundleName);

        if (bundleSizes.ContainsKey(bundleName))
        {
            currentCacheUsage -= bundleSizes[bundleName];
            bundleSizes.Remove(bundleName);
        }

        accessTimes.Remove(bundleName);
    }
}

void UpdateAccessTime(string bundleName)
{
    if (accessTimes.ContainsKey(bundleName))
    {
```

```
            accessTimes[bundleName] = DateTime.Now;
        }
    }

    bool IsPrayerTime()
    {
        if (PrayerTimeSystem.Instance != null)
        {
            return PrayerTimeSystem.Instance.GetNextPrayerProximity() > 0.7f;
        }
        return false;
    }

    bool IsCulturalEvent()
    {
        // Check if currently in cultural event
        return Time.time % 86400 > 54000 && Time.time % 86400 < 57600; //
Simplified: 3 PM - 4 PM
    }

    void ValidateCulturalLoading()
    {
        // Validate that cultural assets are being loaded respectfully
        if (maintainCulturalAuthenticity)
        {
            // Check loading order respects cultural priorities
            // Implementation would validate loading sequence
        }
    }

    public AssetBundleSnapshot GetAssetBundleSnapshot()
```

```
    {
        return new AssetBundleSnapshot
        {
            streaming_enabled = enableStreaming,
            caching_enabled = enableCaching,
            bundles_loaded = loadedBundles.Count,
            bundles_loading = loadingBundles.Count,
            cache_usage_mb = currentCacheUsage,
            cache_limit_mb = maxCacheSize,
            cultural_priority = prioritizeCulturalAssets,
            prayer_time_respect = respectPrayerTimeLoading,
            authenticity_maintained = maintainCulturalAuthenticity
        };
    }

    [System.Serializable]
    public class AssetBundleSnapshot
    {
        public bool streaming_enabled;
        public bool caching_enabled;
        public int bundles_loaded;
        public int bundles_loading;
        public float cache_usage_mb;
        public float cache_limit_mb;
        public bool cultural_priority;
        public bool prayer_time_respect;
        public bool authenticity_maintained;
    }
}
```

45. VersionControlSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;
using System;

[BurstCompile]
public class VersionControlSystem : MonoBehaviour
{
    [BurstCompile]
    struct UpdateValidationSystem : IJobParallelFor
    {
        [ReadOnly] public NativeArray<int> fileVersions;
        [ReadOnly] public NativeArray<int> serverVersions;
        [ReadOnly] public NativeArray<bool> isCritical;
        [ReadOnly] public NativeArray<bool> isCultural;
        [WriteOnly] public NativeArray<bool> updateRequired;
        [WriteOnly] public NativeArray<bool> updateUrgent;
        [WriteOnly] public NativeArray<int> updatePriority;

        public int currentBuildVersion;
        public int minimumSupportedVersion;
        public bool respectCulturalUpdates;
        public float culturalSensitivityWeight;

        public void Execute(int index)
        {
```

```
            int fileVersion = fileVersions[index];
            int serverVersion = serverVersions[index];
            bool critical = isCritical[index];
            bool cultural = isCultural[index];

            // Determine if update is required
            bool required = IsUpdateRequired(fileVersion, serverVersion);
            bool urgent = IsUpdateUrgent(required, critical, cultural);
            int priority = CalculateUpdatePriority(required, critical, cultural, index);

            updateRequired[index] = required;
            updateUrgent[index] = urgent;
            updatePriority[index] = priority;
        }

        [BurstCompile]
        bool IsUpdateRequired(int localVersion, int serverVersion)
        {
            return serverVersion > localVersion && serverVersion >=
minimumSupportedVersion;
        }

        [BurstCompile]
        bool IsUpdateUrgent(bool required, bool critical, bool cultural)
        {
            if (!required) return false;

            if (critical) return true;
            if (cultural && respectCulturalUpdates) return true;

            return false;
```

```
        }

        [BurstCompile]
        int CalculateUpdatePriority(bool required, bool critical, bool cultural, int
index)
        {
            if (!required) return 0;

            int priority = 1;
            if (critical) priority += 5;
            if (cultural && respectCulturalUpdates) priority += 3;

            return math.min(priority, 10);
        }
    }

    [BurstCompile]
    struct MaldivianCulturalUpdateValidation : IJob
    {
        public NativeArray<bool> prayerTimeUpdates;
        public NativeArray<bool> culturalContentUpdates;
        public NativeArray<bool> traditionalPracticeUpdates;
        public NativeArray<bool> religiousContentUpdates;
        public NativeArray<bool> communityEventUpdates;

        public NativeArray<float> culturalSensitivityScores;
        public NativeArray<bool> isRespectfulTiming;
        public NativeArray<int> appropriateUpdateWindows;

        public float currentPrayerTimeWeight;
        public bool isReligiousHoliday;
```

```
        public bool isCulturalFestival;
        public int currentTimeHour;
        public string currentLocale;

        public void Execute()
        {
            ValidatePrayerTimeUpdates();
            ValidateCulturalContentUpdates();
            ValidateTraditionalPracticeUpdates();
            CalculateRespectfulUpdateTiming();
            DetermineAppropriateUpdateWindows();
        }

        [BurstCompile]
        void ValidatePrayerTimeUpdates()
        {
            // Validate updates to prayer time systems
            for (int i = 0; i < prayerTimeUpdates.Length; i++)
            {
                bool isPrayerTime = currentPrayerTimeWeight > 0.7f;
                bool canUpdate = !isPrayerTime && currentTimeHour >= 9 &&
currentTimeHour <= 17; // 9 AM - 5 PM
                prayerTimeUpdates[i] = canUpdate;

                if (canUpdate)
                {
                    culturalSensitivityScores[i] = 0.95f;
                    isRespectfulTiming[i] = true;
                }
            }
        }
```

```
    [BurstCompile]
    void ValidateCulturalContentUpdates()
    {
        // Validate updates to cultural content
        for (int i = 0; i < culturalContentUpdates.Length; i++)
        {
            bool canUpdate = !isReligiousHoliday && !isCulturalFestival;
            culturalContentUpdates[i] = canUpdate;

            if (canUpdate)
            {
                culturalSensitivityScores[prayerTimeUpdates.Length + i] = 0.90f;
                appropriateUpdateWindows[prayerTimeUpdates.Length + i] = 2; //
General hours
            }
        }
    }

    [BurstCompile]
    void ValidateTraditionalPracticeUpdates()
    {
        // Validate updates to traditional practice content
        for (int i = 0; i < traditionalPracticeUpdates.Length; i++)
        {
            bool canUpdate = currentLocale == "dv" || currentLocale == "en"; //
Only update in supported locales
            traditionalPracticeUpdates[i] = canUpdate;

            if (canUpdate)
            {
```

```
                culturalSensitivityScores[prayerTimeUpdates.Length +
culturalContentUpdates.Length + i] = 0.85f;
            }
        }
    }

    [BurstCompile]
    void CalculateRespectfulUpdateTiming()
    {
        // Calculate respectful update timing
        bool isRespectfulHour = currentTimeHour >= 10 && currentTimeHour <=
16; // 10 AM - 4 PM
        bool isRespectfulDay = !isReligiousHoliday && !isCulturalFestival;

        float timingScore = 0.0f;
        if (isRespectfulHour) timingScore += 0.5f;
        if (isRespectfulDay) timingScore += 0.5f;

        for (int i = 0; i < religiousContentUpdates.Length; i++)
        {
            isRespectfulTiming[prayerTimeUpdates.Length +
culturalContentUpdates.Length + traditionalPracticeUpdates.Length + i] =
timingScore > 0.7f;
        }
    }

    [BurstCompile]
    void DetermineAppropriateUpdateWindows()
    {
        // Determine appropriate update windows
        int baseWindow = 1; // General maintenance window
```

```
            if (isReligiousHoliday || isCulturalFestival)
            {
                baseWindow = 4; // Post-festival window
            }
            else if (currentPrayerTimeWeight > 0.5f)
            {
                baseWindow = 3; // Between prayer times
            }
            else if (currentTimeHour < 9 || currentTimeHour > 17)
            {
                baseWindow = 2; // Off-hours
            }

            for (int i = 0; i < communityEventUpdates.Length; i++)
            {
                appropriateUpdateWindows[prayerTimeUpdates.Length +
culturalContentUpdates.Length + traditionalPracticeUpdates.Length +
religiousContentUpdates.Length + i] = baseWindow;
            }
        }
    }

    public static VersionControlSystem Instance { get; private set; }

    [Header("Version Control Settings")]
    public string currentVersion = "1.0.0";
    public int buildNumber = 1;
    public bool enableAutoUpdates = true;
    public bool enableBackgroundDownloads = true;
    public float updateCheckInterval = 3600f; // 1 hour
```

```
public bool forceUpdateCritical = true;

[Header("Cultural Update Respect")]
public bool respectPrayerTimes = true;
public bool avoidReligiousHolidays = true;
public bool considerCulturalFestivals = true;
public bool maintainCulturalAuthenticity = true;
public bool enableRespectfulScheduling = true;

[Header("Update Channels")]
public UpdateChannel currentChannel = UpdateChannel.Stable;
public bool allowBetaUpdates = false;
public bool allowExperimental = false;
public string updateServerURL = "https://updates.raajjevagu.com";

[Header("Rollback Settings")]
public bool enableRollback = true;
public int maxRollbackVersions = 3;
public bool autoRollbackOnFailure = true;

private Dictionary<string, VersionInfo> availableUpdates;
private List<string> downloadedUpdates;
private Queue<string> updateQueue;
private bool isUpdating = false;
private string lastUpdateCheck;
private VersionInfo currentVersionInfo;

public enum UpdateChannel
{
    Stable,
    Beta,
```

```
    Experimental,
    Cultural_Preview
}

public enum UpdateType
{
    Minor = 0,
    Major = 1,
    Critical = 2,
    Cultural = 3,
    Religious = 4,
    Emergency = 5
}

[System.Serializable]
public class VersionInfo
{
    public string version;
    public int build;
    public UpdateType type;
    public bool isCultural;
    public bool isCritical;
    public string[] releaseNotes;
    public string downloadURL;
    public float size;
    public string[] culturalNotes;
    public string releaseDate;
    public string[] supportedLocales;
    public bool requiresRestart;
}
```

```
void Awake()
{
    Instance = this;
    InitializeVersionControl();
}

void InitializeVersionControl()
{
    // Initialize collections
    availableUpdates = new Dictionary<string, VersionInfo>();
    downloadedUpdates = new List<string>();
    updateQueue = new Queue<string>();

    // Parse current version
    currentVersionInfo = new VersionInfo
    {
        version = currentVersion,
        build = buildNumber,
        type = UpdateType.Minor,
        isCultural = false,
        isCritical = false
    };

    // Initialize update validation
    int fileCount = 50;
    var localVersions = new NativeArray<int>(fileCount, Allocator.TempJob);
    var serverVersions = new NativeArray<int>(fileCount, Allocator.TempJob);
    var critical = new NativeArray<bool>(fileCount, Allocator.TempJob);
    var cultural = new NativeArray<bool>(fileCount, Allocator.TempJob);
    var required = new NativeArray<bool>(fileCount, Allocator.TempJob);
    var urgent = new NativeArray<bool>(fileCount, Allocator.TempJob);
```

```
var priorities = new NativeArray<int>(fileCount, Allocator.TempJob);

// Initialize with sample version data
for (int i = 0; i < fileCount; i++)
{
    localVersions[i] = buildNumber;
    serverVersions[i] = buildNumber + UnityEngine.Random.Range(0, 3);
    critical[i] = i < 5; // First 5 are critical
    cultural[i] = i % 3 == 0; // Every third is cultural
}

var validationJob = new UpdateValidationSystem
{
    fileVersions = localVersions,
    serverVersions = serverVersions,
    isCritical = critical,
    isCultural = cultural,
    updateRequired = required,
    updateUrgent = urgent,
    updatePriority = priorities,
    currentBuildVersion = buildNumber,
    minimumSupportedVersion = buildNumber - 5,
    respectCulturalUpdates = maintainCulturalAuthenticity,
    culturalSensitivityWeight = 2.0f
};

// Initialize cultural update validation
var prayerUpdates = new NativeArray<bool>(5, Allocator.TempJob);
var culturalUpdates = new NativeArray<bool>(10, Allocator.TempJob);
var traditionalUpdates = new NativeArray<bool>(8, Allocator.TempJob);
var religiousUpdates = new NativeArray<bool>(3, Allocator.TempJob);
```

```
        var communityUpdates = new NativeArray<bool>(5, Allocator.TempJob);
        var sensitivityScores = new NativeArray<float>(50, Allocator.TempJob);
        var respectfulTiming = new NativeArray<bool>(50, Allocator.TempJob);
        var updateWindows = new NativeArray<int>(50, Allocator.TempJob);

        var culturalJob = new MaldivianCulturalUpdateValidation
        {
            prayerTimeUpdates = prayerUpdates,
            culturalContentUpdates = culturalUpdates,
            traditionalPracticeUpdates = traditionalUpdates,
            religiousContentUpdates = religiousUpdates,
            communityEventUpdates = communityUpdates,
            culturalSensitivityScores = sensitivityScores,
            isRespectfulTiming = respectfulTiming,
            appropriateUpdateWindows = updateWindows,
            currentPrayerTimeWeight = GetPrayerTimeWeight(),
            isReligiousHoliday = IsReligiousHoliday(),
            isCulturalFestival = IsCulturalFestival(),
            currentTimeHour = DateTime.Now.Hour,
            currentLocale = "dv"
        };

        JobHandle validationHandle = validationJob.Schedule(fileCount, 8);
        JobHandle culturalHandle = culturalJob.Schedule(validationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessValidationResults(required, urgent, priorities);
        ProcessCulturalValidation(prayerUpdates, culturalUpdates,
traditionalUpdates, sensitivityScores, respectfulTiming);
```

```
        // Cleanup
        localVersions.Dispose();
        serverVersions.Dispose();
        critical.Dispose();
        cultural.Dispose();
        required.Dispose();
        urgent.Dispose();
        priorities.Dispose();
        prayerUpdates.Dispose();
        culturalUpdates.Dispose();
        traditionalUpdates.Dispose();
        religiousUpdates.Dispose();
        communityUpdates.Dispose();
        sensitivityScores.Dispose();
        respectfulTiming.Dispose();
        updateWindows.Dispose();

        StartUpdateMonitoring();
    }

    void ProcessValidationResults(NativeArray<bool> required, NativeArray<bool>
urgent, NativeArray<int> priorities)
    {
        // Process update validation results
        int requiredCount = 0;
        int urgentCount = 0;
        for (int i = 0; i < required.Length; i++)
        {
            if (required[i]) requiredCount++;
            if (urgent[i]) urgentCount++;
        }
```

```
        Debug.Log($"Update validation: {requiredCount} updates required,
{urgentCount} urgent");
    }

    void ProcessCulturalValidation(NativeArray<bool> prayer, NativeArray<bool>
cultural, NativeArray<bool> traditional, NativeArray<float> sensitivity,
NativeArray<bool> timing)
    {
        // Process cultural validation results
        int respectfulCount = 0;
        for (int i = 0; i < timing.Length; i++)
        {
            if (timing[i]) respectfulCount++;
        }
        Debug.Log($"Cultural validation: {respectfulCount} updates have respectful
timing");
    }

    void StartUpdateMonitoring()
    {
        // Start update monitoring
        if (enableAutoUpdates)
        {
            InvokeRepeating("CheckForUpdates", 0f, updateCheckInterval);
            InvokeRepeating("ValidateCulturalTiming", 0f, 300f); // Every 5 minutes
        }
    }

    void CheckForUpdates()
    {
        if (!enableAutoUpdates || isUpdating) return;
```

```
    StartCoroutine(CheckForUpdatesAsync());
}

System.Collections.IEnumerator CheckForUpdatesAsync()
{
    // Simulate update check (in production, would call server API)
    yield return new WaitForSeconds(1f);

    // Generate sample available updates
    GenerateSampleUpdates();

    lastUpdateCheck = DateTime.Now.ToString();
}

void GenerateSampleUpdates()
{
    // Generate sample updates for demonstration
    VersionInfo update1 = new VersionInfo
    {
        version = "1.1.0",
        build = buildNumber + 1,
        type = UpdateType.Minor,
        isCultural = false,
        isCritical = false,
        size = 25.5f,
        releaseDate = DateTime.Now.AddDays(-1).ToString(),
        requiresRestart = false
    };

    VersionInfo update2 = new VersionInfo
```

```
        {
            version = "1.1.1",
            build = buildNumber + 2,
            type = UpdateType.Cultural,
            isCultural = true,
            isCritical = false,
            size = 15.2f,
            releaseDate = DateTime.Now.AddDays(-0.5).ToString(),
            requiresRestart = false,
            culturalNotes = new string[] { "Updated prayer time calculations", "Added
traditional fishing practices" }
        };

        availableUpdates["1.1.0"] = update1;
        availableUpdates["1.1.1"] = update2;
    }

    public void DownloadUpdate(string version)
    {
        if (!availableUpdates.ContainsKey(version))
        {
            Debug.LogError($"Update not available: {version}");
            return;
        }

        VersionInfo update = availableUpdates[version];

        if (!CanUpdateRespectfully(update))
        {
            Debug.LogWarning($"Update {version} cannot be applied at this time due
to cultural considerations");
```

```
            QueueUpdateForLater(update);
            return;
        }

        StartCoroutine(DownloadUpdateAsync(update));
    }

    bool CanUpdateRespectfully(VersionInfo update)
    {
        if (!enableRespectfulScheduling) return true;

        // Check if update can be applied respectfully
        if (respectPrayerTimes && IsPrayerTime())
        {
            if (update.type != UpdateType.Critical && update.type !=
UpdateType.Emergency)
            {
                return false;
            }
        }

        if (avoidReligiousHolidays && IsReligiousHoliday())
        {
            return false;
        }

        if (considerCulturalFestivals && IsCulturalFestival())
        {
            return false;
        }
```

```
    return true;
}

void QueueUpdateForLater(VersionInfo update)
{
    updateQueue.Enqueue(update.version);
    Debug.Log($"Update {update.version} queued for respectful timing");
}

System.Collections.IEnumerator DownloadUpdateAsync(VersionInfo update)
{
    isUpdating = true;

    float downloadSize = update.size * 1024f * 1024f; // Convert to bytes
    float downloaded = 0f;
    float downloadSpeed = 1f * 1024f * 1024f; // 1 MB/s

    while (downloaded < downloadSize)
    {
        yield return null;

        downloaded += downloadSpeed * Time.deltaTime;
        float progress = downloaded / downloadSize;

        Debug.Log($"Downloading update {update.version}: {progress:P}");

        // Check for respectful timing interruptions
        if (!CanUpdateRespectfully(update))
        {
            Debug.LogWarning("Download interrupted for cultural respect");
            yield break;
```

```
        }
    }

    // Apply update
    ApplyUpdate(update);
}

void ApplyUpdate(VersionInfo update)
{
    // Simulate update application
    Debug.Log($"Applying update {update.version}");

    if (update.requiresRestart)
    {
        Debug.Log("Update requires restart");
        // Schedule restart
    }
    else
    {
        // Apply update immediately
        currentVersion = update.version;
        buildNumber = update.build;
        currentVersionInfo = update;

        downloadedUpdates.Add(update.version);
        availableUpdates.Remove(update.version);
    }

    isUpdating = false;
}
```

```
    public void RollbackToVersion(string version)
    {
        if (!enableRollback)
        {
            Debug.LogWarning("Rollback is not enabled");
            return;
        }

        if (!downloadedUpdates.Contains(version))
        {
            Debug.LogError($"Cannot rollback to {version} - not in download
history");
            return;
        }

        StartCoroutine(RollbackAsync(version));
    }

    System.Collections.IEnumerator RollbackAsync(string version)
    {
        isUpdating = true;

        // Simulate rollback process
        yield return new WaitForSeconds(2f);

        // Apply rollback
        currentVersion = version;
        buildNumber = int.Parse(version.Split('.')[2]);

        Debug.Log($"Rolled back to version {version}");
        isUpdating = false;
```

```
}

void ValidateCulturalTiming()
{
    // Validate that updates respect cultural timing
    if (enableRespectfulScheduling)
    {
        bool isRespectfulTime = IsRespectfulUpdateTime();

        if (!isRespectfulTime && updateQueue.Count > 0)
        {
            // Wait for respectful time
            return;
        }

        if (isRespectfulTime && updateQueue.Count > 0)
        {
            // Process queued updates
            while (updateQueue.Count > 0)
            {
                string version = updateQueue.Dequeue();
                DownloadUpdate(version);
            }
        }
    }
}

bool IsRespectfulUpdateTime()
{
    int hour = DateTime.Now.Hour;
    bool isBusinessHours = hour >= 9 && hour <= 17; // 9 AM - 5 PM
```

```
    bool isPrayerTime = IsPrayerTime();
    bool isHoliday = IsReligiousHoliday() || IsCulturalFestival();

    return isBusinessHours && !isPrayerTime && !isHoliday;
}

float GetPrayerTimeWeight()
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0.0f;
}

bool IsReligiousHoliday()
{
    // Simplified check - would check religious calendar
    return DateTime.Now.DayOfWeek == DayOfWeek.Friday;
}

bool IsCulturalFestival()
{
    // Simplified check - would check cultural calendar
    return DateTime.Now.Day == 1; // First day of month
}

public VersionSnapshot GetVersionSnapshot()
{
    return new VersionSnapshot
    {
```

```
            current_version = currentVersion,
            build_number = buildNumber,
            auto_updates = enableAutoUpdates,
            respectful_scheduling = enableRespectfulScheduling,
            available_updates = availableUpdates.Count,
            queued_updates = updateQueue.Count,
            last_check = lastUpdateCheck,
            cultural_respect = maintainCulturalAuthenticity
        };
    }

    [System.Serializable]
    public class VersionSnapshot
    {
        public string current_version;
        public int build_number;
        public bool auto_updates;
        public bool respectful_scheduling;
        public int available_updates;
        public int queued_updates;
        public string last_check;
        public bool cultural_respect;
    }
}
```

46. MainGameManager.cs

```
using UnityEngine;
```

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;
using System;

[BurstCompile]
public class MainGameManager : MonoBehaviour
{
    [BurstCompile]
    struct GameStateOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> systemPerformanceScores;
        [ReadOnly] public NativeArray<bool> systemActiveStates;
        [ReadOnly] public NativeArray<int> systemPriorities;
        [WriteOnly] public NativeArray<bool> shouldOptimizeSystem;
        [WriteOnly] public NativeArray<float> optimizationScores;
        [WriteOnly] public NativeArray<int> optimizationRecommendations;

        public float targetFrameTime;
        public int maxActiveSystems;
        public float performanceThreshold;

        public void Execute(int index)
        {
            float performance = systemPerformanceScores[index];
            bool active = systemActiveStates[index];
            int priority = systemPriorities[index];
```

```
            bool shouldOptimize = ShouldOptimizeSystem(performance, active,
priority);
            float score = CalculateOptimizationScore(performance, priority);
            int recommendation =
GenerateOptimizationRecommendation(performance, active, priority);

            shouldOptimizeSystem[index] = shouldOptimize;
            optimizationScores[index] = score;
            optimizationRecommendations[index] = recommendation;
        }

        [BurstCompile]
        bool ShouldOptimizeSystem(float performance, bool active, int priority)
        {
            return active && performance < performanceThreshold && priority < 8;
        }

        [BurstCompile]
        float CalculateOptimizationScore(float performance, int priority)
        {
            float baseScore = math.saturate(performance / 0.016f); // 60 FPS
baseline
            float priorityBonus = (10 - priority) * 0.1f;
            return math.saturate(baseScore + priorityBonus);
        }

        [BurstCompile]
        int GenerateOptimizationRecommendation(float performance, bool active,
int priority)
        {
            if (!active) return 0;
```

```
        if (performance < 0.008f) return 3; // Critical optimization
        if (performance < 0.012f) return 2; // Major optimization
        if (performance < 0.016f) return 1; // Minor optimization
        return 0; // No optimization needed
    }
}

[BurstCompile]
struct MaldivianGameSessionValidation : IJob
{
    public NativeArray<bool> prayerTimeRespect;
    public NativeArray<bool> culturalEventHandling;
    public NativeArray<bool> traditionalPracticeIntegration;
    public NativeArray<bool> communityInteractionSupport;
    public NativeArray<bool> environmentalAwareness;

    public NativeArray<float> culturalAuthenticityScores;
    public NativeArray<bool> isCulturallyAppropriate;
    public NativeArray<int> respectfulSessionStates;

    public float currentPrayerTimeWeight;
    public bool isReligiousContext;
    public bool isCommunityEvent;
    public int currentIslandID;
    public float sessionDuration;

    public void Execute()
    {
        ValidatePrayerTimeRespect();
        ValidateCulturalEventHandling();
        ValidateTraditionalPracticeIntegration();
```

```
    CalculateCulturalAppropriateness();
    DetermineRespectfulSessionState();
}

[BurstCompile]
void ValidatePrayerTimeRespect()
{
    for (int i = 0; i < prayerTimeRespect.Length; i++)
    {
        bool shouldRespect = currentPrayerTimeWeight > 0.5f;
        prayerTimeRespect[i] = shouldRespect;

        if (shouldRespect)
        {
            culturalAuthenticityScores[i] = 0.95f;
        }
    }
}

[BurstCompile]
void ValidateCulturalEventHandling()
{
    for (int i = 0; i < culturalEventHandling.Length; i++)
    {
        bool shouldHandle = isCommunityEvent || isReligiousContext;
        culturalEventHandling[i] = shouldHandle;

        if (shouldHandle)
        {
            isCulturallyAppropriate[i] = true;
        }
```

```
    }
}

[BurstCompile]
void ValidateTraditionalPracticeIntegration()
{
    for (int i = 0; i < traditionalPracticeIntegration.Length; i++)
    {
        bool shouldIntegrate = currentIslandID >= 0 && currentIslandID < 41;
        traditionalPracticeIntegration[i] = shouldIntegrate;

        if (shouldIntegrate)
        {
            culturalAuthenticityScores[prayerTimeRespect.Length + i] = 0.90f;
        }
    }
}

[BurstCompile]
void CalculateCulturalAppropriateness()
{
    float totalScore = 0.0f;
    int totalChecks = 0;

    for (int i = 0; i < prayerTimeRespect.Length; i++)
    {
        if (prayerTimeRespect[i]) totalScore += 1.0f;
        totalChecks++;
    }

    for (int i = 0; i < culturalEventHandling.Length; i++)
```

```
        {
            if (culturalEventHandling[i]) totalScore += 1.0f;
            totalChecks++;
        }

        float averageScore = totalChecks > 0 ? totalScore / totalChecks : 0.0f;

        for (int i = 0; i < isCulturallyAppropriate.Length; i++)
        {
            isCulturallyAppropriate[i] = averageScore > 0.8f;
        }
    }

    [BurstCompile]
    void DetermineRespectfulSessionState()
    {
        int state = 1; // Normal gameplay

        if (currentPrayerTimeWeight > 0.7f) state = 2; // Quiet/respectful mode
        if (isCommunityEvent) state = 3; // Community event mode
        if (isReligiousContext) state = 4; // Religious context mode

        for (int i = 0; i < respectfulSessionStates.Length; i++)
        {
            respectfulSessionStates[i] = state;
        }
    }
}

public static MainGameManager Instance { get; private set; }
```

```
[Header("Game State Management")]
public GameState currentState = GameState.MainMenu;
public GameState previousState;
public bool pauseOnFocusLoss = true;
public bool enableAutoSave = true;
public float autoSaveInterval = 300f; // 5 minutes

[Header("Cultural Integration")]
public bool respectPrayerTimes = true;
public bool enableCulturalEvents = true;
public bool maintainAuthenticity = true;
public bool adaptiveDifficulty = true;

[Header("Performance Management")]
public bool autoOptimizePerformance = true;
public int targetFrameRate = 60;
public float performanceCheckInterval = 5f;
public MobileTier targetDeviceTier = MobileTier.MidRange;

[Header("System References")]
public PlayerController playerController;
public CameraSystem cameraSystem;
public AudioManager audioManager;
public UIManager uiManager;
public InputManager inputManager;

private Dictionary<string, GameSystem> gameSystems;
private List<GameState> stateHistory;
private float lastAutoSave;
private float lastPerformanceCheck;
private bool isInitialized = false;
```

```
public enum GameState
{
    MainMenu,
    Loading,
    Playing,
    Paused,
    PrayerTime,
    CulturalEvent,
    CommunityInteraction,
    Settings,
    Quitting
}

public enum MobileTier
{
    LowEnd,
    MidRange,
    HighEnd
}

void Awake()
{
    if (Instance == null)
    {
        Instance = this;
        DontDestroyOnLoad(gameObject);
        InitializeGameManager();
    }
    else
    {
```

```
            Destroy(gameObject);
        }
    }

    void InitializeGameManager()
    {
        // Initialize collections
        gameSystems = new Dictionary<string, GameSystem>();
        stateHistory = new List<GameState>();

        // Initialize performance optimization
        int systemCount = 20;
        var performances = new NativeArray<float>(systemCount,
Allocator.TempJob);
        var activeStates = new NativeArray<bool>(systemCount,
Allocator.TempJob);
        var priorities = new NativeArray<int>(systemCount, Allocator.TempJob);
        var shouldOptimize = new NativeArray<bool>(systemCount,
Allocator.TempJob);
        var scores = new NativeArray<float>(systemCount, Allocator.TempJob);
        var recommendations = new NativeArray<int>(systemCount,
Allocator.TempJob);

        // Initialize with sample system data
        for (int i = 0; i < systemCount; i++)
        {
            performances[i] = UnityEngine.Random.Range(0.008f, 0.018f); // 55-125
FPS range
            activeStates[i] = UnityEngine.Random.Range(0, 2) == 1;
            priorities[i] = UnityEngine.Random.Range(1, 10);
        }
```

```
        var optimizationJob = new GameStateOptimization
        {
            systemPerformanceScores = performances,
            systemActiveStates = activeStates,
            systemPriorities = priorities,
            shouldOptimizeSystem = shouldOptimize,
            optimizationScores = scores,
            optimizationRecommendations = recommendations,
            targetFrameTime = 1.0f / targetFrameRate,
            maxActiveSystems = 15,
            performanceThreshold = 0.016f
        };

        // Initialize cultural session validation
        var prayerRespect = new NativeArray<bool>(5, Allocator.TempJob);
        var culturalHandling = new NativeArray<bool>(8, Allocator.TempJob);
        var traditionalIntegration = new NativeArray<bool>(10, Allocator.TempJob);
        var communitySupport = new NativeArray<bool>(6, Allocator.TempJob);
        var environmentalAwareness = new NativeArray<bool>(4,
Allocator.TempJob);
        var authenticityScores = new NativeArray<float>(50, Allocator.TempJob);
        var culturalAppropriate = new NativeArray<bool>(50, Allocator.TempJob);
        var sessionStates = new NativeArray<int>(50, Allocator.TempJob);

        var culturalJob = new MaldivianGameSessionValidation
        {
            prayerTimeRespect = prayerRespect,
            culturalEventHandling = culturalHandling,
            traditionalPracticeIntegration = traditionalIntegration,
            communityInteractionSupport = communitySupport,
```

```
            environmentalAwareness = environmentalAwareness,
            culturalAuthenticityScores = authenticityScores,
            isCulturallyAppropriate = culturalAppropriate,
            respectfulSessionStates = sessionStates,
            currentPrayerTimeWeight = GetPrayerTimeWeight(),
            isReligiousContext = IsReligiousContext(),
            isCommunityEvent = IsCommunityEvent(),
            currentIslandID = GetCurrentIslandID(),
            sessionDuration = 0f
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(systemCount,
8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessOptimizationResults(shouldOptimize, scores, recommendations);
        ProcessCulturalValidation(prayerRespect, culturalHandling,
authenticityScores, sessionStates);

        // Cleanup
        performances.Dispose();
        activeStates.Dispose();
        priorities.Dispose();
        shouldOptimize.Dispose();
        scores.Dispose();
        recommendations.Dispose();
        prayerRespect.Dispose();
        culturalHandling.Dispose();
        traditionalIntegration.Dispose();
```

```
        communitySupport.Dispose();
        environmentalAwareness.Dispose();
        authenticityScores.Dispose();
        culturalAppropriate.Dispose();
        sessionStates.Dispose();

        RegisterGameSystems();
        StartGameMonitoring();

        isInitialized = true;
        Debug.Log("Main Game Manager initialized successfully");
    }

    void ProcessOptimizationResults(NativeArray<bool> shouldOptimize,
NativeArray<float> scores, NativeArray<int> recommendations)
    {
        int optimizeCount = 0;
        float avgScore = 0f;

        for (int i = 0; i < shouldOptimize.Length; i++)
        {
            if (shouldOptimize[i]) optimizeCount++;
            avgScore += scores[i];
        }

        avgScore /= shouldOptimize.Length;

        Debug.Log($"Game state optimization: {optimizeCount} systems need
optimization, average score: {avgScore:F2}");
    }
```

```
    void ProcessCulturalValidation(NativeArray<bool> prayer, NativeArray<bool>
cultural, NativeArray<float> authenticity, NativeArray<int> states)
    {
        int respectfulCount = 0;
        for (int i = 0; i < prayer.Length; i++)
        {
            if (prayer[i]) respectfulCount++;
        }

        int culturalCount = 0;
        for (int i = 0; i < cultural.Length; i++)
        {
            if (cultural[i]) culturalCount++;
        }

        Debug.Log($"Cultural validation: {respectfulCount} prayer respects,
{culturalCount} cultural handlers");
    }

    void RegisterGameSystems()
    {
        // Register all core game systems
        RegisterSystem("Player", FindObjectOfType<PlayerController>());
        RegisterSystem("Camera", FindObjectOfType<CameraSystem>());
        RegisterSystem("Audio", FindObjectOfType<AudioManager>());
        RegisterSystem("UI", FindObjectOfType<UIManager>());
        RegisterSystem("Input", FindObjectOfType<InputManager>());
        RegisterSystem("Scene", FindObjectOfType<GameSceneManager>());
        RegisterSystem("Assets", FindObjectOfType<AssetManager>());
        RegisterSystem("Weather", FindObjectOfType<WeatherSystem>());
        RegisterSystem("Prayer", FindObjectOfType<PrayerTimeSystem>());
```

```
    RegisterSystem("Fishing", FindObjectOfType<FishingSystem>());
}

void RegisterSystem(string name, GameSystem system)
{
    if (system != null)
    {
        gameSystems[name] = system;
        system.Initialize(this);
    }
}

void StartGameMonitoring()
{
    InvokeRepeating("CheckPerformance", 1f, performanceCheckInterval);
    InvokeRepeating("AutoSaveGame", 30f, autoSaveInterval);
    InvokeRepeating("ValidateCulturalState", 10f, 60f);
}

void Update()
{
    if (!isInitialized) return;

    // Handle state transitions
    HandleStateTransitions();

    // Check for cultural events
    CheckCulturalEvents();

    // Auto-save check
    if (enableAutoSave && Time.time - lastAutoSave > autoSaveInterval)
```

```
    {
        SaveGameState();
    }
}

void HandleStateTransitions()
{
    // Handle transitions between game states
    if (currentState == GameState.Playing)
    {
        // Check for prayer time transition
        if (respectPrayerTimes && IsPrayerTime())
        {
            ChangeState(GameState.PrayerTime);
        }

        // Check for cultural events
        if (enableCulturalEvents && IsCulturalEvent())
        {
            ChangeState(GameState.CulturalEvent);
        }
    }
}

void CheckCulturalEvents()
{
    // Monitor for cultural events that should affect game state
    if (maintainAuthenticity)
    {
        float prayerWeight = GetPrayerTimeWeight();
        if (prayerWeight > 0.7f && currentState == GameState.Playing)
```

```
        {
            // Transition to respectful mode
            SetRespectfulMode(true);
        }
        else if (prayerWeight < 0.3f && previousState == GameState.PrayerTime)
        {
            // Return to normal mode
            SetRespectfulMode(false);
        }
    }
}

public void ChangeState(GameState newState)
{
    if (currentState == newState) return;

    previousState = currentState;
    currentState = newState;
    stateHistory.Add(newState);

    // Notify all systems of state change
    foreach (var system in gameSystems.Values)
    {
        system.OnStateChanged(currentState, previousState);
    }

    // Handle state-specific logic
    OnStateChanged(newState, previousState);
}

void OnStateChanged(GameState newState, GameState oldState)
```

```
{
    switch (newState)
    {
        case GameState.PrayerTime:
            Time.timeScale = 0.5f; // Slow down time during prayer
            audioManager?.SetRespectfulMode(true);
            uiManager?.ShowPrayerNotification();
            break;

        case GameState.CulturalEvent:
            Time.timeScale = 1f;
            audioManager?.SetCulturalMode(true);
            uiManager?.ShowCulturalEventUI();
            break;

        case GameState.Playing:
            Time.timeScale = 1f;
            audioManager?.SetRespectfulMode(false);
            audioManager?.SetCulturalMode(false);
            break;

        case GameState.Paused:
            Time.timeScale = 0f;
            break;
    }
}

public void SetRespectfulMode(bool enabled)
{
    if (enabled)
    {
```

```
        audioManager?.SetRespectfulMode(true);
        uiManager?.ShowRespectfulUI();
    }
    else
    {
        audioManager?.SetRespectfulMode(false);
        uiManager?.HideRespectfulUI();
    }
}

public void SaveGameState()
{
    // Save current game state
    GameSaveData saveData = new GameSaveData
    {
        currentState = currentState,
        playerPosition = playerController?.transform.position ?? Vector3.zero,
        currentIsland = GetCurrentIslandID(),
        gameTime = Time.time,
        prayerTimeWeight = GetPrayerTimeWeight(),
        culturalEventActive = IsCulturalEvent()
    };

    // Save to persistent storage
    string saveJson = JsonUtility.ToJson(saveData);
    PlayerPrefs.SetString("RVA_GameState", saveJson);
    PlayerPrefs.Save();

    lastAutoSave = Time.time;
    Debug.Log("Game state saved");
}
```

```
    public void LoadGameState()
    {
        // Load saved game state
        string saveJson = PlayerPrefs.GetString("RVA_GameState", "");
        if (!string.IsNullOrEmpty(saveJson))
        {
            GameSaveData saveData =
JsonUtility.FromJson<GameSaveData>(saveJson);

            // Restore game state
            ChangeState(saveData.currentState);

            // Restore player position
            if (playerController != null)
            {
                playerController.transform.position = saveData.playerPosition;
            }

            Debug.Log("Game state loaded");
        }
    }

    void CheckPerformance()
    {
        if (!autoOptimizePerformance) return;

        float currentFPS = 1.0f / Time.unscaledDeltaTime;

        if (currentFPS < targetFrameRate * 0.8f)
        {
```

```
        // Performance is below target, optimize
        OptimizePerformance();
    }
}

void OptimizePerformance()
{
    // Implement performance optimization logic
    Debug.Log("Optimizing performance...");

    // Could reduce draw distance, lower quality settings, etc.
    QualitySettings.SetQualityLevel(1, true);
}

public T GetSystem<T>() where T : GameSystem
{
    foreach (var system in gameSystems.Values)
    {
        if (system is T typedSystem)
        {
            return typedSystem;
        }
    }
    return null;
}

float GetPrayerTimeWeight()
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
```

```
        }
        return 0f;
    }

    bool IsPrayerTime()
    {
        return GetPrayerTimeWeight() > 0.7f;
    }

    bool IsCulturalEvent()
    {
        // Simplified check - would check cultural calendar
        return Time.time % 86400 > 43200 && Time.time % 86400 < 46800; // 12
PM - 1 PM
    }

    bool IsReligiousContext()
    {
        return GetPrayerTimeWeight() > 0.5f;
    }

    int GetCurrentIslandID()
    {
        // Get current island ID from world system
        return 0; // Placeholder
    }

    void AutoSaveGame()
    {
        if (enableAutoSave && currentState == GameState.Playing)
        {
```

```
            SaveGameState();
        }
    }

    void ValidateCulturalState()
    {
        // Validate that game state respects cultural requirements
        if (maintainAuthenticity)
        {
            bool isRespectful = !IsPrayerTime() || currentState ==
GameState.PrayerTime;

            if (!isRespectful)
            {
                Debug.LogWarning("Game state may not be culturally respectful");
                ChangeState(GameState.PrayerTime);
            }
        }
    }

    void OnApplicationPause(bool pauseStatus)
    {
        if (pauseOnFocusLoss)
        {
            if (pauseStatus)
            {
                ChangeState(GameState.Paused);
            }
            else if (currentState == GameState.Paused)
            {
                ChangeState(previousState);
```

```
        }
    }
}

void OnApplicationFocus(bool hasFocus)
{
    if (pauseOnFocusLoss)
    {
        if (!hasFocus)
        {
            ChangeState(GameState.Paused);
        }
        else if (currentState == GameState.Paused)
        {
            ChangeState(previousState);
        }
    }
}

public GameSnapshot GetGameSnapshot()
{
    return new GameSnapshot
    {
        current_state = currentState,
        fps = 1.0f / Time.unscaledDeltaTime,
        session_time = Time.time,
        prayer_time_weight = GetPrayerTimeWeight(),
        is_cultural_event = IsCulturalEvent(),
        system_count = gameSystems.Count,
        respectful_mode = IsPrayerTime()
    };
```

```
    }

    [System.Serializable]
    public class GameSaveData
    {
        public GameState currentState;
        public Vector3 playerPosition;
        public int currentIsland;
        public float gameTime;
        public float prayerTimeWeight;
        public bool culturalEventActive;
    }

    [System.Serializable]
    public class GameSnapshot
    {
        public GameState current_state;
        public float fps;
        public float session_time;
        public float prayer_time_weight;
        public bool is_cultural_event;
        public int system_count;
        public bool respectful_mode;
    }
}

public abstract class GameSystem : MonoBehaviour
{
    protected MainGameManager gameManager;

    public virtual void Initialize(MainGameManager manager)
```

```
    {
        gameManager = manager;
    }

    public virtual void OnStateChanged(GameState newState, GameState
oldState)
    {
        // Override in derived classes
    }
}
```

47. GameSceneManager.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;
using System.Collections;
using UnityEngine.SceneManagement;

[BurstCompile]
public class GameSceneManager : GameSystem
{
    [BurstCompile]
    struct SceneLoadingOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> sceneSizes;
        [ReadOnly] public NativeArray<int> scenePriorities;
        [ReadOnly] public NativeArray<bool> isEssential;
```

```
    [ReadOnly] public NativeArray<float> loadingTimes;
    [WriteOnly] public NativeArray<bool> shouldPreload;
    [WriteOnly] public NativeArray<int> loadingOrder;
    [WriteOnly] public NativeArray<float> optimizationScores;

    public float availableMemory;
    public float bandwidthLimit;
    public int maxConcurrentLoads;
    public float targetLoadTime;

    public void Execute(int index)
    {
        float size = sceneSizes[index];
        int priority = scenePriorities[index];
        bool essential = isEssential[index];
        float loadTime = loadingTimes[index];

        bool shouldLoad = ShouldPreloadScene(size, priority, essential,
loadTime);
        int order = CalculateLoadingOrder(priority, essential, loadTime);
        float score = CalculateOptimizationScore(size, priority, loadTime);

        shouldPreload[index] = shouldLoad;
        loadingOrder[index] = order;
        optimizationScores[index] = score;
    }

    [BurstCompile]
    bool ShouldPreloadScene(float size, int priority, bool essential, float
loadTime)
    {
```

```
        bool sizeOk = size < availableMemory * 0.1f; // 10% of available memory
        bool priorityOk = priority > 5 || essential;
        bool loadTimeOk = loadTime < targetLoadTime * 2f;

        return sizeOk && priorityOk && loadTimeOk;
    }

    [BurstCompile]
    int CalculateLoadingOrder(int priority, bool essential, float loadTime)
    {
        int order = priority * 100;
        if (essential) order -= 1000;
        order += Mathf.RoundToInt(loadTime * 10f);

        return math.max(0, order);
    }

    [BurstCompile]
    float CalculateOptimizationScore(float size, int priority, float loadTime)
    {
        float sizeScore = math.saturate(1.0f - (size / 100f)); // 100MB baseline
        float priorityScore = priority / 10f;
        float loadTimeScore = math.saturate(1.0f - (loadTime / targetLoadTime));

        return (sizeScore + priorityScore + loadTimeScore) * 0.33f;
    }
}

[BurstCompile]
struct MaldivianSceneValidation : IJob
{
```

```
public NativeArray<bool> islandSceneAuthenticity;
public NativeArray<bool> culturalSceneAccuracy;
public NativeArray<bool> religiousSceneRespect;
public NativeArray<bool> traditionalPracticeScenes;
public NativeArray<bool> communityEventScenes;

public NativeArray<float> culturalSensitivityScores;
public NativeArray<bool> isRespectfulLoading;
public NativeArray<int> appropriateLoadingTimes;

public int currentIslandID;
public bool isPrayerTime;
public bool isCulturalEvent;
public bool isReligiousHoliday;
public float prayerTimeWeight;

public void Execute()
{
    ValidateIslandSceneAuthenticity();
    ValidateCulturalSceneAccuracy();
    ValidateReligiousSceneRespect();
    CalculateRespectfulLoadingTimes();
    DetermineAppropriateLoadingWindows();
}

[BurstCompile]
void ValidateIslandSceneAuthenticity()
{
    for (int i = 0; i < islandSceneAuthenticity.Length; i++)
    {
```

```
            bool isAuthentic = i == currentIslandID && currentIslandID >= 0 &&
currentIslandID < 41;
            islandSceneAuthenticity[i] = isAuthentic;

            if (isAuthentic)
            {
                culturalSensitivityScores[i] = 0.95f;
            }
        }
    }

    [BurstCompile]
    void ValidateCulturalSceneAccuracy()
    {
        for (int i = 0; i < culturalSceneAccuracy.Length; i++)
        {
            bool isAccurate = !isPrayerTime && !isReligiousHoliday;
            culturalSceneAccuracy[i] = isAccurate;

            if (isAccurate)
            {
                isRespectfulLoading[i] = true;
            }
        }
    }

    [BurstCompile]
    void ValidateReligiousSceneRespect()
    {
        for (int i = 0; i < religiousSceneRespect.Length; i++)
        {
```

```
            bool isRespectful = !isPrayerTime && prayerTimeWeight < 0.5f;
            religiousSceneRespect[i] = isRespectful;

            if (isRespectful)
            {
                appropriateLoadingTimes[i] = 1; // General loading time
            }
            else
            {
                appropriateLoadingTimes[i] = 3; // Between prayer times
            }
        }
    }

    [BurstCompile]
    void CalculateRespectfulLoadingTimes()
    {
        int baseTime = 1;

        if (isPrayerTime) baseTime = 4; // Post-prayer
        else if (isCulturalEvent) baseTime = 2; // Cultural event time
        else if (isReligiousHoliday) baseTime = 5; // Post-holiday

        for (int i = 0; i < traditionalPracticeScenes.Length; i++)
        {
            appropriateLoadingTimes[islandSceneAuthenticity.Length +
culturalSceneAccuracy.Length + i] = baseTime;
        }
    }

    [BurstCompile]
```

```
    void DetermineAppropriateLoadingWindows()
    {
        for (int i = 0; i < communityEventScenes.Length; i++)
        {
            bool canLoad = !isPrayerTime && !isReligiousHoliday;
            communityEventScenes[i] = canLoad;

            if (canLoad)
            {
                appropriateLoadingTimes[islandSceneAuthenticity.Length +
culturalSceneAccuracy.Length + religiousSceneRespect.Length + i] = 1;
            }
            else
            {
                appropriateLoadingTimes[islandSceneAuthenticity.Length +
culturalSceneAccuracy.Length + religiousSceneRespect.Length + i] = 4;
            }
        }
    }
}

public static GameSceneManager Instance { get; private set; }

[Header("Scene Configuration")]
public string mainMenuScene = "MainMenu";
public string[] islandScenes = new string[41];
public string[] culturalScenes = new string[10];
public string loadingScene = "LoadingScreen";

[Header("Loading Settings")]
public bool enableAsyncLoading = true;
```

```
public bool showLoadingScreen = true;
public float minimumLoadingTime = 2f;
public int maxConcurrentLoads = 2;
public bool respectPrayerTimes = true;

[Header("Cultural Settings")]
public bool maintainCulturalAuthenticity = true;
public bool respectfulSceneLoading = true;
public bool prioritizeCulturalScenes = true;

private Dictionary<string, SceneLoadRequest> loadRequests;
private Queue<SceneLoadRequest> loadQueue;
private List<AsyncOperation> activeLoads;
private SceneLoadRequest currentLoad;
private bool isLoading = false;

public enum LoadPriority
{
    Critical = 0,
    High = 1,
    Medium = 2,
    Low = 3,
    Cultural = 4,
    Background = 5
}

public class SceneLoadRequest
{
    public string sceneName;
    public LoadPriority priority;
    public bool showLoadingScreen;
```

```
        public bool isCultural;
        public bool isEssential;
        public Action<float> onProgress;
        public Action<string> onComplete;
        public Action<string> onError;
        public float estimatedSize;
        public float estimatedLoadTime;
    }

    public override void Initialize(MainGameManager manager)
    {
        base.Initialize(manager);
        InitializeSceneManager();
    }

    void InitializeSceneManager()
    {
        // Initialize collections
        loadRequests = new Dictionary<string, SceneLoadRequest>();
        loadQueue = new Queue<SceneLoadRequest>();
        activeLoads = new List<AsyncOperation>();

        // Initialize scene loading optimization
        int sceneCount = 60;
        var sizes = new NativeArray<float>(sceneCount, Allocator.TempJob);
        var priorities = new NativeArray<int>(sceneCount, Allocator.TempJob);
        var essential = new NativeArray<bool>(sceneCount, Allocator.TempJob);
        var loadTimes = new NativeArray<float>(sceneCount, Allocator.TempJob);
        var shouldPreload = new NativeArray<bool>(sceneCount,
Allocator.TempJob);
        var order = new NativeArray<int>(sceneCount, Allocator.TempJob);
```

```
var scores = new NativeArray<float>(sceneCount, Allocator.TempJob);

// Initialize with sample scene data
for (int i = 0; i < sceneCount; i++)
{
    sizes[i] = UnityEngine.Random.Range(10f, 100f); // 10-100 MB
    priorities[i] = UnityEngine.Random.Range(1, 10);
    essential[i] = i < 15; // First 15 are essential
    loadTimes[i] = UnityEngine.Random.Range(1f, 5f); // 1-5 seconds
}

var optimizationJob = new SceneLoadingOptimization
{
    sceneSizes = sizes,
    scenePriorities = priorities,
    isEssential = essential,
    loadingTimes = loadTimes,
    shouldPreload = shouldPreload,
    loadingOrder = order,
    optimizationScores = scores,
    availableMemory = SystemInfo.systemMemorySize,
    bandwidthLimit = 10f * 1024f * 1024f, // 10 MB/s
    maxConcurrentLoads = maxConcurrentLoads,
    targetLoadTime = minimumLoadingTime
};

// Initialize cultural scene validation
var islandScenes = new NativeArray<bool>(41, Allocator.TempJob);
var culturalScenesArray = new NativeArray<bool>(10, Allocator.TempJob);
var religiousScenes = new NativeArray<bool>(5, Allocator.TempJob);
var traditionalScenes = new NativeArray<bool>(8, Allocator.TempJob);
```

```
        var communityScenes = new NativeArray<bool>(6, Allocator.TempJob);
        var sensitivityScores = new NativeArray<float>(70, Allocator.TempJob);
        var respectfulLoading = new NativeArray<bool>(70, Allocator.TempJob);
        var loadingTimes = new NativeArray<int>(70, Allocator.TempJob);

        var culturalJob = new MaldivianSceneValidation
        {
            islandSceneAuthenticity = islandScenes,
            culturalSceneAccuracy = culturalScenesArray,
            religiousSceneRespect = religiousScenes,
            traditionalPracticeScenes = traditionalScenes,
            communityEventScenes = communityScenes,
            culturalSensitivityScores = sensitivityScores,
            isRespectfulLoading = respectfulLoading,
            appropriateLoadingTimes = loadingTimes,
            currentIslandID = GetCurrentIslandID(),
            isPrayerTime = IsPrayerTime(),
            isCulturalEvent = IsCulturalEvent(),
            isReligiousHoliday = IsReligiousHoliday(),
            prayerTimeWeight = GetPrayerTimeWeight()
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(sceneCount, 8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessOptimizationResults(shouldPreload, order, scores);
        ProcessCulturalValidation(islandScenes, culturalScenesArray,
religiousScenes, sensitivityScores, loadingTimes);
```

```
        // Cleanup
        sizes.Dispose();
        priorities.Dispose();
        essential.Dispose();
        loadTimes.Dispose();
        shouldPreload.Dispose();
        order.Dispose();
        scores.Dispose();
        islandScenes.Dispose();
        culturalScenesArray.Dispose();
        religiousScenes.Dispose();
        traditionalScenes.Dispose();
        communityScenes.Dispose();
        sensitivityScores.Dispose();
        respectfulLoading.Dispose();
        loadingTimes.Dispose();

        StartSceneMonitoring();
    }

    void ProcessOptimizationResults(NativeArray<bool> shouldPreload,
NativeArray<int> order, NativeArray<float> scores)
    {
        int preloadCount = 0;
        float avgScore = 0f;

        for (int i = 0; i < shouldPreload.Length; i++)
        {
            if (shouldPreload[i]) preloadCount++;
            avgScore += scores[i];
        }
```

```
        avgScore /= shouldPreload.Length;

        Debug.Log($"Scene optimization: {preloadCount} scenes should be preloaded, average score: {avgScore:F2}");
    }

    void ProcessCulturalValidation(NativeArray<bool> island, NativeArray<bool> cultural, NativeArray<bool> religious, NativeArray<float> sensitivity, NativeArray<int> times)
    {
        int authenticCount = 0;
        for (int i = 0; i < island.Length; i++)
        {
            if (island[i]) authenticCount++;
        }

        int respectfulCount = 0;
        for (int i = 0; i < religious.Length; i++)
        {
            if (religious[i]) respectfulCount++;
        }

        Debug.Log($"Cultural validation: {authenticCount} authentic island scenes, {respectfulCount} respectful religious scenes");
    }

    void StartSceneMonitoring()
    {
        InvokeRepeating("ProcessLoadQueue", 0.5f, 0.1f);
        InvokeRepeating("MonitorActiveLoads", 0f, 0.05f);
```

```
        InvokeRepeating("ValidateCulturalLoading", 10f, 30f);
    }

    public void LoadScene(string sceneName, LoadPriority priority =
LoadPriority.Medium, bool showLoading = true, Action<string> onComplete =
null)
    {
        SceneLoadRequest request = new SceneLoadRequest
        {
            sceneName = sceneName,
            priority = priority,
            showLoadingScreen = showLoading && showLoadingScreen,
            isCultural = IsCulturalScene(sceneName),
            isEssential = priority == LoadPriority.Critical,
            onComplete = onComplete,
            estimatedSize = EstimateSceneSize(sceneName),
            estimatedLoadTime = EstimateLoadTime(sceneName)
        };

        // Check if we can load respectfully
        if (respectfulSceneLoading && !CanLoadRespectfully(request))
        {
            QueueLoadForLater(request);
            return;
        }

        QueueLoadRequest(request);
    }

    bool CanLoadRespectfully(SceneLoadRequest request)
    {
```

```
        if (!respectPrayerTimes) return true;

        // Check if it's prayer time
        if (IsPrayerTime() && !request.isEssential)
        {
            return false;
        }

        // Check if it's a religious holiday
        if (IsReligiousHoliday() && request.isCultural)
        {
            return false;
        }

        return true;
    }

    void QueueLoadForLater(SceneLoadRequest request)
    {
        // Queue the load for a more respectful time
        StartCoroutine(WaitForRespectfulTime(request));
    }

    System.Collections.IEnumerator WaitForRespectfulTime(SceneLoadRequest
request)
    {
        while (!CanLoadRespectfully(request))
        {
            yield return new WaitForSeconds(1f);
        }
```

```
        QueueLoadRequest(request);
    }

    void QueueLoadRequest(SceneLoadRequest request)
    {
        loadRequests[request.sceneName] = request;

        // Insert into queue based on priority
        List<SceneLoadRequest> tempList = new
List<SceneLoadRequest>(loadQueue.ToArray());
        tempList.Add(request);
        tempList.Sort((a, b) => ((int)a.priority).CompareTo((int)b.priority));

        loadQueue.Clear();
        foreach (var item in tempList)
        {
            loadQueue.Enqueue(item);
        }
    }

    void ProcessLoadQueue()
    {
        if (isLoading || loadQueue.Count == 0) return;
        if (activeLoads.Count >= maxConcurrentLoads) return;

        currentLoad = loadQueue.Dequeue();

        if (currentLoad.showLoadingScreen)
        {
            ShowLoadingScreen();
        }
```

```
        StartCoroutine(LoadSceneAsync(currentLoad));
    }

    System.Collections.IEnumerator LoadSceneAsync(SceneLoadRequest
request)
    {
        isLoading = true;

        // Simulate minimum loading time for smooth experience
        float startTime = Time.time;

        // Load loading scene if needed
        if (request.showLoadingScreen && !string.IsNullOrEmpty(loadingScene))
        {
            AsyncOperation loadingOp =
SceneManager.LoadSceneAsync(loadingScene, LoadSceneMode.Additive);
            yield return loadingOp;
        }

        // Load the actual scene
        AsyncOperation loadOp =
SceneManager.LoadSceneAsync(request.sceneName);
        activeLoads.Add(loadOp);

        while (!loadOp.isDone)
        {
            float progress = loadOp.progress;
            request.onProgress?.Invoke(progress);

            // Update loading screen
```

```
            if (request.showLoadingScreen)
            {
                UpdateLoadingScreen(progress);
            }

            yield return null;
        }

        // Ensure minimum loading time
        float elapsedTime = Time.time - startTime;
        if (elapsedTime < minimumLoadingTime)
        {
            yield return new WaitForSeconds(minimumLoadingTime - elapsedTime);
        }

        // Clean up loading scene
        if (request.showLoadingScreen && !string.IsNullOrEmpty(loadingScene))
        {
            SceneManager.UnloadSceneAsync(loadingScene);
        }

        activeLoads.Remove(loadOp);
        isLoading = false;

        request.onComplete?.Invoke(request.sceneName);

        // Load next scene in queue
        ProcessLoadQueue();
    }

    void ShowLoadingScreen()
```

```
    {
        // Show loading screen UI
        if (gameManager?.GetSystem<UIManager>() != null)
        {
            gameManager.GetSystem<UIManager>().ShowLoadingScreen();
        }
    }

    void UpdateLoadingScreen(float progress)
    {
        // Update loading progress
        if (gameManager?.GetSystem<UIManager>() != null)
        {

gameManager.GetSystem<UIManager>().UpdateLoadingProgress(progress);
        }
    }

    void MonitorActiveLoads()
    {
        // Monitor and clean up completed loads
        for (int i = activeLoads.Count - 1; i >= 0; i--)
        {
            if (activeLoads[i].isDone)
            {
                activeLoads.RemoveAt(i);
            }
        }
    }

    public void UnloadScene(string sceneName)
```

```
{
    if (SceneManager.GetSceneByName(sceneName).isLoaded)
    {
        SceneManager.UnloadSceneAsync(sceneName);
    }
}

bool IsCulturalScene(string sceneName)
{
    // Check if scene contains cultural content
    return sceneName.Contains("Cultural") ||
           sceneName.Contains("Traditional") ||
           sceneName.Contains("Religious") ||
           sceneName.Contains("Mosque") ||
           sceneName.Contains("Community");
}

float EstimateSceneSize(string sceneName)
{
    // Estimate scene size based on name/content
    if (IsCulturalScene(sceneName))
    {
        return UnityEngine.Random.Range(20f, 50f); // Cultural scenes are larger
    }

    if (sceneName.Contains("Island"))
    {
        return UnityEngine.Random.Range(30f, 80f); // Island scenes are largest
    }

    return UnityEngine.Random.Range(10f, 30f); // Standard scenes
```

```
}

float EstimateLoadTime(string sceneName)
{
    float size = EstimateSceneSize(sceneName);
    return size / 20f; // Assume 20 MB/s loading speed
}

public void PreloadScenes(string[] sceneNames)
{
    foreach (string sceneName in sceneNames)
    {
        LoadScene(sceneName, LoadPriority.Background, false);
    }
}

void ValidateCulturalLoading()
{
    // Validate that scene loading respects cultural requirements
    if (maintainCulturalAuthenticity)
    {
        bool isRespectfulTime = !IsPrayerTime() && !IsReligiousHoliday();

        if (!isRespectfulTime && loadQueue.Count > 0)
        {
            Debug.Log("Scene loading paused for cultural respect");
        }
    }
}

float GetPrayerTimeWeight()
```

```
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0f;
}

bool IsPrayerTime()
{
    return GetPrayerTimeWeight() > 0.7f;
}

bool IsCulturalEvent()
{
    // Simplified check
    return Time.time % 86400 > 43200 && Time.time % 86400 < 46800;
}

bool IsReligiousHoliday()
{
    // Simplified check - would check religious calendar
    return DateTime.Now.DayOfWeek == DayOfWeek.Friday;
}

int GetCurrentIslandID()
{
    // Get current island ID from game state
    return 0; // Placeholder
}
```

```
    public override void OnStateChanged(GameState newState, GameState
oldState)
    {
        base.OnStateChanged(newState, oldState);

        switch (newState)
        {
            case GameState.Loading:
                // Pause non-essential loading during prayer times
                if (IsPrayerTime() && respectfulSceneLoading)
                {
                    // Defer non-essential loads
                }
                break;

            case GameState.PrayerTime:
                // Pause all non-essential scene loading
                if (respectPrayerTimes)
                {
                    // Clear non-essential loads from queue
                }
                break;
        }
    }

    public SceneSnapshot GetSceneSnapshot()
    {
        return new SceneSnapshot
        {
            is_loading = isLoading,
            queue_size = loadQueue.Count,
```

```
            active_loads = activeLoads.Count,
            current_scene = currentLoad?.sceneName ?? "None",
            cultural_respect = respectfulSceneLoading,
            prayer_time_active = IsPrayerTime()
        };
    }

    [System.Serializable]
    public class SceneSnapshot
    {
        public bool is_loading;
        public int queue_size;
        public int active_loads;
        public string current_scene;
        public bool cultural_respect;
        public bool prayer_time_active;
    }
}
```

48. PlayerController.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;

[BurstCompile]
public class PlayerController : GameSystem
```

```
{
    [BurstCompile]
    struct MovementOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> movementInputs;
        [ReadOnly] public NativeArray<float> deltaTimes;
        [ReadOnly] public NativeArray<float3> currentPositions;
        [ReadOnly] public NativeArray<quaternion> currentRotations;
        [WriteOnly] public NativeArray<float3> newPositions;
        [WriteOnly] public NativeArray<float3> newVelocities;
        [WriteOnly] public NativeArray<bool> collisionResults;

        public float moveSpeed;
        public float rotationSpeed;
        public float3 worldBounds;
        public float deltaTime;

        public void Execute(int index)
        {
            float3 input = movementInputs[index];
            float3 currentPos = currentPositions[index];
            quaternion currentRot = currentRotations[index];

            // Calculate movement
            float3 movement = CalculateMovement(input, currentRot);
            float3 newPos = currentPos + movement * moveSpeed * deltaTime;

            // Validate position
            bool validPosition = ValidatePosition(newPos, worldBounds);

            // Apply collision detection
```

```
    bool hasCollision = CheckCollision(newPos);

    if (validPosition && !hasCollision)
    {
        newPositions[index] = newPos;
        newVelocities[index] = movement * moveSpeed;
    }
    else
    {
        newPositions[index] = currentPos;
        newVelocities[index] = float3.zero;
    }

    collisionResults[index] = hasCollision;
}

[BurstCompile]
float3 CalculateMovement(float3 input, quaternion rotation)
{
    float3 forward = math.mul(rotation, new float3(0, 0, 1));
    float3 right = math.mul(rotation, new float3(1, 0, 0));

    return (forward * input.z + right * input.x) * math.length(input);
}

[BurstCompile]
bool ValidatePosition(float3 position, float3 bounds)
{
    return math.abs(position.x) <= bounds.x * 0.5f &&
           math.abs(position.z) <= bounds.z * 0.5f;
}
```

```
    [BurstCompile]
    bool CheckCollision(float3 position)
    {
        // Simplified collision check
        return false; // Would check against collision world
    }
}

[BurstCompile]
struct MaldivianCulturalMovement : IJob
{
    public NativeArray<bool> prayerMovementRestrictions;
    public NativeArray<bool> culturalAreaRespect;
    public NativeArray<bool> traditionalPracticeZones;
    public NativeArray<bool> communitySpaceAwareness;
    public NativeArray<bool> religiousSiteProximity;

    public NativeArray<float> culturalSensitivityScores;
    public NativeArray<bool> isMovementAppropriate;
    public NativeArray<int> appropriateMovementModes;

    public float3 currentPosition;
    public float3 targetPosition;
    public float prayerTimeWeight;
    public bool isReligiousContext;
    public bool isCommunityInteraction;
    public int currentIslandID;

    public void Execute()
    {
```

```
    ValidatePrayerMovementRestrictions();
    ValidateCulturalAreaRespect();
    ValidateTraditionalPracticeZones();
    CalculateMovementAppropriateness();
    DetermineAppropriateMovementMode();
}

[BurstCompile]
void ValidatePrayerMovementRestrictions()
{
    for (int i = 0; i < prayerMovementRestrictions.Length; i++)
    {
        bool shouldRestrict = prayerTimeWeight > 0.7f;
        prayerMovementRestrictions[i] = shouldRestrict;

        if (shouldRestrict)
        {
            culturalSensitivityScores[i] = 0.95f;
        }
    }
}

[BurstCompile]
void ValidateCulturalAreaRespect()
{
    for (int i = 0; i < culturalAreaRespect.Length; i++)
    {
        bool shouldRespect = isReligiousContext || isCommunityInteraction;
        culturalAreaRespect[i] = shouldRespect;

        if (shouldRespect)
```

```
            {
                isMovementAppropriate[i] = true;
            }
        }
    }

    [BurstCompile]
    void ValidateTraditionalPracticeZones()
    {
        for (int i = 0; i < traditionalPracticeZones.Length; i++)
        {
            bool isInZone = currentIslandID >= 0 && currentIslandID < 41;
            traditionalPracticeZones[i] = isInZone;

            if (isInZone)
            {
                culturalSensitivityScores[prayerMovementRestrictions.Length + i] =
0.90f;
            }
        }
    }

    [BurstCompile]
    void CalculateMovementAppropriateness()
    {
        float totalScore = 0.0f;
        int totalChecks = 0;

        for (int i = 0; i < prayerMovementRestrictions.Length; i++)
        {
```

```
            if (!prayerMovementRestrictions[i]) totalScore += 1.0f; // Not restricted
= good
            totalChecks++;
        }

        for (int i = 0; i < culturalAreaRespect.Length; i++)
        {
            if (culturalAreaRespect[i]) totalScore += 1.0f;
            totalChecks++;
        }

        float averageScore = totalChecks > 0 ? totalScore / totalChecks : 0.0f;

        for (int i = 0; i < isMovementAppropriate.Length; i++)
        {
            isMovementAppropriate[i] = averageScore > 0.7f;
        }
    }

    [BurstCompile]
    void DetermineAppropriateMovementMode()
    {
        int mode = 1; // Normal movement

        if (prayerTimeWeight > 0.7f) mode = 2; // Restricted movement
        if (isReligiousContext) mode = 3; // Respectful movement
        if (isCommunityInteraction) mode = 4; // Community-aware movement

        for (int i = 0; i < appropriateMovementModes.Length; i++)
        {
            appropriateMovementModes[i] = mode;
```

```
        }
    }
}

public static PlayerController Instance { get; private set; }

[Header("Movement Settings")]
public float walkSpeed = 5f;
public float runSpeed = 8f;
public float rotationSpeed = 720f;
public float jumpForce = 8f;
public float gravity = -20f;
public bool enableInertia = true;
public float inertiaDamping = 5f;

[Header("Cultural Movement")]
public bool respectPrayerTimes = true;
public bool enableCulturalMovement = true;
public bool maintainRespectfulDistance = true;
public float respectfulSpeedMultiplier = 0.5f;

[Header("Mobile Optimization")]
public bool optimizeForMobile = true;
public bool useBurstCompilation = true;
public int physicsIterations = 1;
public bool reduceMovementPrecision = false;

private CharacterController characterController;
private Animator animator;
private Transform cameraTransform;
private Vector3 currentVelocity;
```

```
private Vector3 movementInput;
private bool isGrounded;
private bool isRunning;
private bool isPrayerTime;
private float currentSpeed;
private Vector3 lastPosition;

public enum MovementMode
{
    Normal = 0,
    Respectful = 1,
    Prayer = 2,
    Community = 3,
    Restricted = 4
}

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

public override void Initialize(MainGameManager manager)
{
```

```
        base.Initialize(manager);
        InitializePlayerController();
    }

    void InitializePlayerController()
    {
        // Get components
        characterController = GetComponent<CharacterController>();
        animator = GetComponent<Animator>();
        cameraTransform = Camera.main?.transform;

        // Initialize movement optimization
        int sampleCount = 100;
        var inputs = new NativeArray<float3>(sampleCount, Allocator.TempJob);
        var deltaTimes = new NativeArray<float>(sampleCount,
Allocator.TempJob);
        var positions = new NativeArray<float3>(sampleCount, Allocator.TempJob);
        var rotations = new NativeArray<quaternion>(sampleCount,
Allocator.TempJob);
        var newPositions = new NativeArray<float3>(sampleCount,
Allocator.TempJob);
        var velocities = new NativeArray<float3>(sampleCount, Allocator.TempJob);
        var collisions = new NativeArray<bool>(sampleCount, Allocator.TempJob);

        // Initialize with sample movement data
        for (int i = 0; i < sampleCount; i++)
        {
            inputs[i] = UnityEngine.Random.insideUnitSphere;
            deltaTimes[i] = Time.deltaTime;
            positions[i] = transform.position + UnityEngine.Random.insideUnitSphere
* 10f;
```

```
            rotations[i] =
quaternion.LookRotation(UnityEngine.Random.insideUnitSphere, Vector3.up);
        }

        var movementJob = new MovementOptimization
        {
            movementInputs = inputs,
            deltaTimes = deltaTimes,
            currentPositions = positions,
            currentRotations = rotations,
            newPositions = newPositions,
            newVelocities = velocities,
            collisionResults = collisions,
            moveSpeed = walkSpeed,
            rotationSpeed = rotationSpeed,
            worldBounds = new float3(1000f, 100f, 1000f),
            deltaTime = Time.deltaTime
        };

        // Initialize cultural movement validation
        var prayerRestrictions = new NativeArray<bool>(5, Allocator.TempJob);
        var culturalRespect = new NativeArray<bool>(8, Allocator.TempJob);
        var traditionalZones = new NativeArray<bool>(10, Allocator.TempJob);
        var communityAwareness = new NativeArray<bool>(6, Allocator.TempJob);
        var religiousProximity = new NativeArray<bool>(4, Allocator.TempJob);
        var sensitivityScores = new NativeArray<float>(50, Allocator.TempJob);
        var movementAppropriate = new NativeArray<bool>(50,
Allocator.TempJob);
        var movementModes = new NativeArray<int>(50, Allocator.TempJob);

        var culturalJob = new MaldivianCulturalMovement
```

```
        {
            prayerMovementRestrictions = prayerRestrictions,
            culturalAreaRespect = culturalRespect,
            traditionalPracticeZones = traditionalZones,
            communitySpaceAwareness = communityAwareness,
            religiousSiteProximity = religiousProximity,
            culturalSensitivityScores = sensitivityScores,
            isMovementAppropriate = movementAppropriate,
            appropriateMovementModes = movementModes,
            currentPosition = transform.position,
            targetPosition = transform.position + Vector3.forward * 10f,
            prayerTimeWeight = GetPrayerTimeWeight(),
            isReligiousContext = IsReligiousContext(),
            isCommunityInteraction = IsCommunityInteraction(),
            currentIslandID = GetCurrentIslandID()
        };

        JobHandle movementHandle = movementJob.Schedule(sampleCount, 8);
        JobHandle culturalHandle = culturalJob.Schedule(movementHandle);
        culturalHandle.Complete();

        // Process results
        ProcessMovementResults(newPositions, velocities, collisions);
        ProcessCulturalValidation(prayerRestrictions, culturalRespect,
sensitivityScores, movementModes);

        // Cleanup
        inputs.Dispose();
        deltaTimes.Dispose();
        positions.Dispose();
        rotations.Dispose();
```

```
        newPositions.Dispose();
        velocities.Dispose();
        collisions.Dispose();
        prayerRestrictions.Dispose();
        culturalRespect.Dispose();
        traditionalZones.Dispose();
        communityAwareness.Dispose();
        religiousProximity.Dispose();
        sensitivityScores.Dispose();
        movementAppropriate.Dispose();
        movementModes.Dispose();

        currentSpeed = walkSpeed;
        lastPosition = transform.position;
    }

    void ProcessMovementResults(NativeArray<float3> positions,
NativeArray<float3> velocities, NativeArray<bool> collisions)
    {
        int validMovements = 0;
        int collisionCount = 0;

        for (int i = 0; i < positions.Length; i++)
        {
            if (math.length(velocities[i]) > 0.1f) validMovements++;
            if (collisions[i]) collisionCount++;
        }

        Debug.Log($"Movement optimization: {validMovements} valid movements,
{collisionCount} collisions detected");
    }
```

```
    void ProcessCulturalValidation(NativeArray<bool> prayer, NativeArray<bool>
cultural, NativeArray<float> sensitivity, NativeArray<int> modes)
    {
        int restrictedCount = 0;
        for (int i = 0; i < prayer.Length; i++)
        {
            if (prayer[i]) restrictedCount++;
        }

        int respectfulCount = 0;
        for (int i = 0; i < cultural.Length; i++)
        {
            if (cultural[i]) respectfulCount++;
        }

        Debug.Log($"Cultural validation: {restrictedCount} prayer restrictions,
{respectfulCount} cultural respects");
    }

    void Update()
    {
        if (!gameManager) return;

        HandleInput();
        UpdateMovement();
        UpdateAnimation();
        CheckCulturalContext();
    }

    void HandleInput()
```

```
    {
        // Get input from InputManager
        if (gameManager.GetSystem<InputManager>() != null)
        {
            var inputManager = gameManager.GetSystem<InputManager>();
            movementInput = inputManager.GetMovementInput();
            isRunning = inputManager.IsRunning();
        }
        else
        {
            // Fallback to direct input
            movementInput = new Vector3(Input.GetAxis("Horizontal"), 0,
Input.GetAxis("Vertical"));
            isRunning = Input.GetKey(KeyCode.LeftShift);
        }

        // Normalize input
        if (movementInput.magnitude > 1f)
        {
            movementInput.Normalize();
        }
    }

    void UpdateMovement()
    {
        // Check cultural movement restrictions
        MovementMode mode = GetMovementMode();

        // Apply movement restrictions
        float speed = GetSpeedForMode(mode);
        bool canMove = CanMoveInCurrentContext(mode);
```

```
        if (canMove && movementInput.magnitude > 0.1f)
        {
            // Calculate movement direction
            Vector3 movement = CalculateMovementDirection(movementInput);

            // Apply movement
            Vector3 velocity = movement * speed;

            if (enableInertia)
            {
                currentVelocity = Vector3.Lerp(currentVelocity, velocity,
inertiaDamping * Time.deltaTime);
            }
            else
            {
                currentVelocity = velocity;
            }

            // Apply gravity
            ApplyGravity();

            // Move character
            characterController.Move(currentVelocity * Time.deltaTime);

            // Rotate character
            if (movementInput.magnitude > 0.1f)
            {
                RotateCharacter(movement);
            }
        }
```

```
        else
        {
            // Decelerate
            currentVelocity = Vector3.Lerp(currentVelocity, Vector3.zero,
inertiaDamping * Time.deltaTime);
            characterController.Move(currentVelocity * Time.deltaTime);
        }

        // Update grounded state
        isGrounded = characterController.isGrounded;
    }

    Vector3 CalculateMovementDirection(Vector3 input)
    {
        if (cameraTransform != null)
        {
            // Camera-relative movement
            Vector3 forward = cameraTransform.forward;
            Vector3 right = cameraTransform.right;

            forward.y = 0f;
            right.y = 0f;

            forward.Normalize();
            right.Normalize();

            return (forward * input.z + right * input.x).normalized;
        }
        else
        {
            // World-relative movement
```

```
            return new Vector3(input.x, 0f, input.z).normalized;
        }
    }

    void RotateCharacter(Vector3 direction)
    {
        if (direction.magnitude > 0.1f)
        {
            Quaternion targetRotation = Quaternion.LookRotation(direction);
            transform.rotation = Quaternion.RotateTowards(transform.rotation,
targetRotation, rotationSpeed * Time.deltaTime);
        }
    }

    void ApplyGravity()
    {
        if (isGrounded && currentVelocity.y < 0)
        {
            currentVelocity.y = -0.5f;
        }
        else
        {
            currentVelocity.y += gravity * Time.deltaTime;
        }
    }

    MovementMode GetMovementMode()
    {
        if (respectPrayerTimes && IsPrayerTime())
        {
            return MovementMode.Prayer;
```

```
        }

        if (IsReligiousContext())
        {
            return MovementMode.Respectful;
        }

        if (IsCommunityInteraction())
        {
            return MovementMode.Community;
        }

        return MovementMode.Normal;
    }

    float GetSpeedForMode(MovementMode mode)
    {
        float baseSpeed = isRunning ? runSpeed : walkSpeed;

        switch (mode)
        {
            case MovementMode.Prayer:
                return baseSpeed * respectfulSpeedMultiplier * 0.3f; // Very slow during
prayer

            case MovementMode.Respectful:
                return baseSpeed * respectfulSpeedMultiplier;

            case MovementMode.Community:
                return baseSpeed * 0.8f; // Slightly slower in community areas
```

```
            case MovementMode.Restricted:
                return baseSpeed * 0.2f; // Very restricted movement

            default:
                return baseSpeed;
        }
    }

    bool CanMoveInCurrentContext(MovementMode mode)
    {
        switch (mode)
        {
            case MovementMode.Prayer:
                // Very limited movement during prayer time
                return movementInput.magnitude < 0.3f; // Only allow very small
movements

            case MovementMode.Restricted:
                // No movement in restricted mode
                return false;

            default:
                return true;
        }
    }

    void UpdateAnimation()
    {
        if (animator != null)
        {
            float speed = currentVelocity.magnitude;
```

```
        animator.SetFloat("Speed", speed);
        animator.SetBool("IsRunning", isRunning && speed > walkSpeed * 0.8f);
        animator.SetBool("IsGrounded", isGrounded);
    }
}

void CheckCulturalContext()
{
    // Update cultural context
    isPrayerTime = IsPrayerTime();

    // Check if player entered/exited cultural areas
    if (maintainRespectfulDistance)
    {
        // Would check proximity to religious sites, community areas, etc.
    }
}

public void SetMovementMode(MovementMode mode)
{
    // Force a specific movement mode (used by game manager)
    // This overrides the automatic mode detection
}

public void EnableRespectfulMode(bool enabled)
{
    respectPrayerTimes = enabled;
}

public void TeleportTo(Vector3 position)
```

```
    {
        characterController.enabled = false;
        transform.position = position;
        characterController.enabled = true;
        currentVelocity = Vector3.zero;
    }

    float GetPrayerTimeWeight()
    {
        if (PrayerTimeSystem.Instance != null)
        {
            return PrayerTimeSystem.Instance.GetNextPrayerProximity();
        }
        return 0f;
    }

    bool IsPrayerTime()
    {
        return GetPrayerTimeWeight() > 0.7f;
    }

    bool IsReligiousContext()
    {
        return GetPrayerTimeWeight() > 0.5f;
    }

    bool IsCommunityInteraction()
    {
        // Simplified check
        return Vector3.Distance(transform.position, Vector3.zero) < 50f; // Near
community center
```

```
}

int GetCurrentIslandID()
{
    // Get current island ID from world system
    return 0; // Placeholder
}

public MovementSnapshot GetMovementSnapshot()
{
    return new MovementSnapshot
    {
        position = transform.position,
        velocity = currentVelocity,
        is_grounded = isGrounded,
        is_running = isRunning,
        movement_mode = GetMovementMode().ToString(),
        speed = currentVelocity.magnitude,
        prayer_time_restricted = isPrayerTime
    };
}

[System.Serializable]
public class MovementSnapshot
{
    public Vector3 position;
    public Vector3 velocity;
    public bool is_grounded;
    public bool is_running;
    public string movement_mode;
    public float speed;
```

```
        public bool prayer_time_restricted;
    }
}
```

49. CameraSystem.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;

[BurstCompile]
public class CameraSystem : GameSystem
{
    [BurstCompile]
    struct CameraOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float3> targetPositions;
        [ReadOnly] public NativeArray<float3> cameraPositions;
        [ReadOnly] public NativeArray<quaternion> cameraRotations;
        [ReadOnly] public NativeArray<float> zoomLevels;
        [WriteOnly] public NativeArray<float3> optimalPositions;
        [WriteOnly] public NativeArray<quaternion> optimalRotations;
        [WriteOnly] public NativeArray<bool> occlusionResults;

        public float3 worldBounds;
        public float minDistance;
```

```
    public float maxDistance;
    public float occlusionThreshold;

    public void Execute(int index)
    {
        float3 targetPos = targetPositions[index];
        float3 currentPos = cameraPositions[index];
        quaternion currentRot = cameraRotations[index];
        float zoom = zoomLevels[index];

        // Calculate optimal camera position
        float3 optimalPos = CalculateOptimalPosition(targetPos, currentPos,
zoom);
        quaternion optimalRot = CalculateOptimalRotation(targetPos,
optimalPos);

        // Check occlusion
        bool hasOcclusion = CheckOcclusion(targetPos, optimalPos);

        // Adjust for occlusion
        if (hasOcclusion)
        {
            optimalPos = AdjustForOcclusion(targetPos, optimalPos);
        }

        optimalPositions[index] = optimalPos;
        optimalRotations[index] = optimalRot;
        occlusionResults[index] = hasOcclusion;
    }

    [BurstCompile]
```

```
float3 CalculateOptimalPosition(float3 target, float3 current, float zoom)
{
    float3 direction = math.normalize(current - target);
    float distance = math.length(current - target) * zoom;
    distance = math.clamp(distance, minDistance, maxDistance);

    return target + direction * distance;
}

[BurstCompile]
quaternion CalculateOptimalRotation(float3 target, float3 camera)
{
    float3 direction = math.normalize(target - camera);
    return quaternion.LookRotation(direction, math.up());
}

[BurstCompile]
bool CheckOcclusion(float3 target, float3 camera)
{
    float3 direction = target - camera;
    float distance = math.length(direction);

    // Simplified occlusion check
    return distance > occlusionThreshold;
}

[BurstCompile]
float3 AdjustForOcclusion(float3 target, float3 camera)
{
    // Move camera closer to avoid occlusion
    float3 direction = math.normalize(camera - target);
```

```
        return target + direction * (math.length(camera - target) * 0.8f);
    }
}

[BurstCompile]
struct MaldivianCulturalCamera : IJob
{
    public NativeArray<bool> prayerCameraRestrictions;
    public NativeArray<bool> culturalViewRespect;
    public NativeArray<bool> traditionalPracticeViews;
    public NativeArray<bool> communityEventAngles;
    public NativeArray<bool> religiousSiteVisibility;

    public NativeArray<float> culturalSensitivityScores;
    public NativeArray<bool> isCameraAngleRespectful;
    public NativeArray<int> appropriateCameraModes;

    public float3 cameraPosition;
    public float3 targetPosition;
    public float prayerTimeWeight;
    public bool isReligiousContext;
    public bool isCommunityInteraction;
    public int currentIslandID;

    public void Execute()
    {
        ValidatePrayerCameraRestrictions();
        ValidateCulturalViewRespect();
        ValidateTraditionalPracticeViews();
        CalculateCameraAngleRespectfulness();
        DetermineAppropriateCameraMode();
```

```
}

[BurstCompile]
void ValidatePrayerCameraRestrictions()
{
    for (int i = 0; i < prayerCameraRestrictions.Length; i++)
    {
        bool shouldRestrict = prayerTimeWeight > 0.7f;
        prayerCameraRestrictions[i] = shouldRestrict;

        if (shouldRestrict)
        {
            culturalSensitivityScores[i] = 0.95f;
        }
    }
}

[BurstCompile]
void ValidateCulturalViewRespect()
{
    for (int i = 0; i < culturalViewRespect.Length; i++)
    {
        bool shouldRespect = isReligiousContext || isCommunityInteraction;
        culturalViewRespect[i] = shouldRespect;

        if (shouldRespect)
        {
            isCameraAngleRespectful[i] = true;
        }
    }
}
```

```
    [BurstCompile]
    void ValidateTraditionalPracticeViews()
    {
        for (int i = 0; i < traditionalPracticeViews.Length; i++)
        {
            bool isTraditionalView = currentIslandID >= 0 && currentIslandID < 41;
            traditionalPracticeViews[i] = isTraditionalView;

            if (isTraditionalView)
            {
                culturalSensitivityScores[prayerCameraRestrictions.Length + i] =
0.90f;
            }
        }
    }

    [BurstCompile]
    void CalculateCameraAngleRespectfulness()
    {
        float3 cameraDirection = math.normalize(targetPosition -
cameraPosition);
        float3 respectfulDirection = new float3(0, -0.3f, 1); // Slightly downward,
respectful angle

        float angleDifference = math.length(cameraDirection -
respectfulDirection);
        bool isRespectful = angleDifference < 0.5f; // Within reasonable tolerance

        for (int i = 0; i < isCameraAngleRespectful.Length; i++)
        {
```

```
            isCameraAngleRespectful[i] = isRespectful;
        }
    }

    [BurstCompile]
    void DetermineAppropriateCameraMode()
    {
        int mode = 1; // Normal camera

        if (prayerTimeWeight > 0.7f) mode = 2; // Restricted camera
        if (isReligiousContext) mode = 3; // Respectful camera
        if (isCommunityInteraction) mode = 4; // Community-aware camera

        for (int i = 0; i < appropriateCameraModes.Length; i++)
        {
            appropriateCameraModes[i] = mode;
        }
    }
}

public static CameraSystem Instance { get; private set; }

[Header("Camera Target")]
public Transform target;
public Vector3 targetOffset = new Vector3(0, 1.5f, 0);
public bool autoFindTarget = true;

[Header("Camera Settings")]
public float followDistance = 10f;
public float followHeight = 5f;
public float rotationSpeed = 5f;
```

```
public float heightDamping = 2f;
public float rotationDamping = 3f;
public bool enableSmoothing = true;

[Header("Cultural Camera")]
public bool respectPrayerTimes = true;
public bool maintainRespectfulAngles = true;
public bool limitMovementDuringPrayer = true;
public float prayerTimeZoomMultiplier = 1.2f;

[Header("Mobile Optimization")]
public bool optimizeForMobile = true;
public bool reduceUpdateFrequency = false;
public int updateEveryNFrames = 1;
public bool simplifiedOcclusion = true;

private Camera mainCamera;
private Vector3 currentVelocity;
private Vector3 desiredPosition;
private Quaternion desiredRotation;
private float currentZoom = 1f;
private int frameCounter = 0;
private bool isPrayerTime;
private MovementMode currentMode;

public enum MovementMode
{
    Normal = 0,
    Follow = 1,
    Orbit = 2,
    Fixed = 3,
```

```
    Prayer = 4,
    Respectful = 5
}

void Awake()
{
    if (Instance == null)
    {
        Instance = this;
    }
    else
    {
        Destroy(gameObject);
    }
}

public override void Initialize(MainGameManager manager)
{
    base.Initialize(manager);
    InitializeCameraSystem();
}

void InitializeCameraSystem()
{
    mainCamera = GetComponent<Camera>();
    if (mainCamera == null)
    {
        mainCamera = gameObject.AddComponent<Camera>();
    }

    // Initialize camera optimization
```

```
        int sampleCount = 50;
        var targetPositions = new NativeArray<float3>(sampleCount,
Allocator.TempJob);
        var cameraPositions = new NativeArray<float3>(sampleCount,
Allocator.TempJob);
        var cameraRotations = new NativeArray<quaternion>(sampleCount,
Allocator.TempJob);
        var zoomLevels = new NativeArray<float>(sampleCount,
Allocator.TempJob);
        var optimalPositions = new NativeArray<float3>(sampleCount,
Allocator.TempJob);
        var optimalRotations = new NativeArray<quaternion>(sampleCount,
Allocator.TempJob);
        var occlusions = new NativeArray<bool>(sampleCount, Allocator.TempJob);

        // Initialize with sample camera data
        for (int i = 0; i < sampleCount; i++)
        {
            targetPositions[i] = UnityEngine.Random.insideUnitSphere * 50f;
            cameraPositions[i] = targetPositions[i] +
UnityEngine.Random.insideUnitSphere * 15f;
            cameraRotations[i] =
quaternion.LookRotation(UnityEngine.Random.insideUnitSphere, Vector3.up);
            zoomLevels[i] = UnityEngine.Random.Range(0.5f, 2f);
        }

        var optimizationJob = new CameraOptimization
        {
            targetPositions = targetPositions,
            cameraPositions = cameraPositions,
            cameraRotations = cameraRotations,
```

```
    zoomLevels = zoomLevels,
    optimalPositions = optimalPositions,
    optimalRotations = optimalRotations,
    occlusionResults = occlusions,
    worldBounds = new float3(2000f, 500f, 2000f),
    minDistance = 2f,
    maxDistance = 50f,
    occlusionThreshold = 25f
};

// Initialize cultural camera validation
var prayerRestrictions = new NativeArray<bool>(6, Allocator.TempJob);
var culturalRespect = new NativeArray<bool>(8, Allocator.TempJob);
var traditionalViews = new NativeArray<bool>(10, Allocator.TempJob);
var communityAngles = new NativeArray<bool>(5, Allocator.TempJob);
var religiousVisibility = new NativeArray<bool>(3, Allocator.TempJob);
var sensitivityScores = new NativeArray<float>(50, Allocator.TempJob);
var respectfulAngles = new NativeArray<bool>(50, Allocator.TempJob);
var cameraModes = new NativeArray<int>(50, Allocator.TempJob);

var culturalJob = new MaldivianCulturalCamera
{
    prayerCameraRestrictions = prayerRestrictions,
    culturalViewRespect = culturalRespect,
    traditionalPracticeViews = traditionalViews,
    communityEventAngles = communityAngles,
    religiousSiteVisibility = religiousVisibility,
    culturalSensitivityScores = sensitivityScores,
    isCameraAngleRespectful = respectfulAngles,
    appropriateCameraModes = cameraModes,
    cameraPosition = transform.position,
```

```
            targetPosition = target ? target.position : Vector3.zero,
            prayerTimeWeight = GetPrayerTimeWeight(),
            isReligiousContext = IsReligiousContext(),
            isCommunityInteraction = IsCommunityInteraction(),
            currentIslandID = GetCurrentIslandID()
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(sampleCount,
8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessOptimizationResults(optimalPositions, optimalRotations,
occlusions);
        ProcessCulturalValidation(prayerRestrictions, culturalRespect,
sensitivityScores, cameraModes);

        // Cleanup
        targetPositions.Dispose();
        cameraPositions.Dispose();
        cameraRotations.Dispose();
        zoomLevels.Dispose();
        optimalPositions.Dispose();
        optimalRotations.Dispose();
        occlusions.Dispose();
        prayerRestrictions.Dispose();
        culturalRespect.Dispose();
        traditionalViews.Dispose();
        communityAngles.Dispose();
        religiousVisibility.Dispose();
```

```
        sensitivityScores.Dispose();
        respectfulAngles.Dispose();
        cameraModes.Dispose();

        // Find target if not assigned
        if (target == null && autoFindTarget)
        {
            FindTarget();
        }

        StartCameraMonitoring();
    }

    void ProcessOptimizationResults(NativeArray<float3> positions,
NativeArray<quaternion> rotations, NativeArray<bool> occlusions)
    {
        int occlusionCount = 0;
        for (int i = 0; i < occlusions.Length; i++)
        {
            if (occlusions[i]) occlusionCount++;
        }

        Debug.Log($"Camera optimization: {occlusionCount} occlusion adjustments
needed");
    }

    void ProcessCulturalValidation(NativeArray<bool> prayer, NativeArray<bool>
cultural, NativeArray<float> sensitivity, NativeArray<int> modes)
    {
        int restrictedCount = 0;
        for (int i = 0; i < prayer.Length; i++)
```

```
        {
            if (prayer[i]) restrictedCount++;
        }

        int respectfulCount = 0;
        for (int i = 0; i < cultural.Length; i++)
        {
            if (cultural[i]) respectfulCount++;
        }

        Debug.Log($"Cultural validation: {restrictedCount} prayer restrictions,
{respectfulCount} cultural respects");
    }

    void StartCameraMonitoring()
    {
        InvokeRepeating("UpdateCameraPosition", 0f, 0.02f); // 50 FPS
        InvokeRepeating("CheckCulturalContext", 1f, 5f);
        InvokeRepeating("ValidateCameraAngles", 2f, 10f);
    }

    void Update()
    {
        if (!gameManager) return;

        // Mobile optimization - skip frames if needed
        if (optimizeForMobile && reduceUpdateFrequency)
        {
            frameCounter++;
            if (frameCounter % updateEveryNFrames != 0)
            {
```

```
            return;
        }
    }

    HandleInput();
    CheckCulturalContext();
}

void LateUpdate()
{
    if (target == null) return;

    UpdateCameraPosition();
    ValidateCameraAngles();
}

void HandleInput()
{
    // Handle camera input from InputManager
    if (gameManager.GetSystem<InputManager>() != null)
    {
        var inputManager = gameManager.GetSystem<InputManager>();

        if (inputManager.IsCameraInputActive())
        {
            Vector2 cameraInput = inputManager.GetCameraInput();

            // Rotate around target
            if (currentMode == MovementMode.Orbit)
            {
                OrbitCamera(cameraInput);
```

```
            }

            // Zoom
            float zoomInput = inputManager.GetZoomInput();
            if (Mathf.Abs(zoomInput) > 0.1f)
            {
                AdjustZoom(zoomInput);
            }
        }
    }
}

void UpdateCameraPosition()
{
    if (target == null) return;

    // Calculate desired position based on mode
    Vector3 targetPosition = target.position + targetOffset;

    switch (GetCurrentMode())
    {
        case MovementMode.Follow:
            UpdateFollowCamera(targetPosition);
            break;

        case MovementMode.Orbit:
            UpdateOrbitCamera(targetPosition);
            break;

        case MovementMode.Fixed:
            UpdateFixedCamera(targetPosition);
```

```
            break;

        case MovementMode.Prayer:
            UpdatePrayerCamera(targetPosition);
            break;

        case MovementMode.Respectful:
            UpdateRespectfulCamera(targetPosition);
            break;

        default:
            UpdateFollowCamera(targetPosition);
            break;
    }

    // Apply position and rotation
    if (enableSmoothing)
    {
        transform.position = Vector3.SmoothDamp(transform.position,
desiredPosition, ref currentVelocity, 0.1f);
        transform.rotation = Quaternion.Slerp(transform.rotation, desiredRotation,
rotationSpeed * Time.deltaTime);
    }
    else
    {
        transform.position = desiredPosition;
        transform.rotation = desiredRotation;
    }
}

void UpdateFollowCamera(Vector3 targetPosition)
```

```
    {
        // Calculate desired position behind and above target
        Vector3 behindDirection = -target.forward;
        if (behindDirection.magnitude < 0.1f)
        {
            behindDirection = Vector3.back;
        }

        desiredPosition = targetPosition + behindDirection * followDistance *
currentZoom;
        desiredPosition += Vector3.up * followHeight * currentZoom;

        // Look at target
        desiredRotation = Quaternion.LookRotation(targetPosition -
transform.position);
    }

    void UpdateOrbitCamera(Vector3 targetPosition)
    {
        // Orbit around target
        float horizontalAngle = Time.time * 10f;
        float verticalAngle = 20f;

        Vector3 orbitPosition = new Vector3(
            Mathf.Cos(horizontalAngle * Mathf.Deg2Rad) * followDistance *
currentZoom,
            Mathf.Sin(verticalAngle * Mathf.Deg2Rad) * followHeight * currentZoom,
            Mathf.Sin(horizontalAngle * Mathf.Deg2Rad) * followDistance *
currentZoom
        );
```

```
        desiredPosition = targetPosition + orbitPosition;
        desiredRotation = Quaternion.LookRotation(targetPosition -
desiredPosition);
    }

    void UpdateFixedCamera(Vector3 targetPosition)
    {
        // Fixed position camera
        desiredPosition = new Vector3(0, followHeight * 2f, -followDistance * 2f);
        desiredRotation = Quaternion.LookRotation(targetPosition -
desiredPosition);
    }

    void UpdatePrayerCamera(Vector3 targetPosition)
    {
        // Respectful camera during prayer time
        Vector3 respectfulOffset = new Vector3(0, followHeight * 0.5f,
followDistance * prayerTimeZoomMultiplier);
        desiredPosition = targetPosition + respectfulOffset;

        // Gentle, respectful angle
        Vector3 lookDirection = targetPosition - desiredPosition;
        lookDirection.y = math.max(lookDirection.y, -0.3f); // Don't look down too
much
        desiredRotation = Quaternion.LookRotation(lookDirection);
    }

    void UpdateRespectfulCamera(Vector3 targetPosition)
    {
        // Respectful camera for cultural areas
```

```
        Vector3 respectfulOffset = new Vector3(0, followHeight * 0.8f,
followDistance * 1.1f);
        desiredPosition = targetPosition + respectfulOffset;
        desiredRotation = Quaternion.LookRotation(targetPosition -
desiredPosition);
    }

    void OrbitCamera(Vector2 input)
    {
        // Implement orbit camera logic
        float rotationAmount = input.x * rotationSpeed * Time.deltaTime;

        Vector3 currentOffset = transform.position - (target.position + targetOffset);
        currentOffset = Quaternion.AngleAxis(rotationAmount, Vector3.up) *
currentOffset;

        desiredPosition = target.position + targetOffset + currentOffset;
        desiredRotation = Quaternion.LookRotation(target.position + targetOffset -
desiredPosition);
    }

    void AdjustZoom(float input)
    {
        float zoomSpeed = 0.1f;
        currentZoom = Mathf.Clamp(currentZoom + input * zoomSpeed, 0.5f, 2f);
    }

    MovementMode GetCurrentMode()
    {
        if (respectPrayerTimes && IsPrayerTime())
        {
```

```
        return MovementMode.Prayer;
    }

    if (maintainRespectfulAngles && IsReligiousContext())
    {
        return MovementMode.Respectful;
    }

    return currentMode;
}

void CheckCulturalContext()
{
    // Update cultural context
    isPrayerTime = IsPrayerTime();

    // Adjust camera settings based on cultural context
    if (isPrayerTime && limitMovementDuringPrayer)
    {
        // Limit camera movement during prayer
        rotationSpeed = 2f; // Slower rotation
    }
    else
    {
        rotationSpeed = 5f; // Normal rotation
    }
}

void ValidateCameraAngles()
{
    // Validate that camera angles are culturally respectful
```

```
        if (maintainRespectfulAngles && IsReligiousContext())
        {
            Vector3 lookDirection = transform.forward;

            // Check if looking down too much (disrespectful)
            if (lookDirection.y < -0.5f)
            {
                // Adjust to more respectful angle
                lookDirection.y = -0.3f;
                desiredRotation = Quaternion.LookRotation(lookDirection);
            }
        }
    }

    void FindTarget()
    {
        // Find player or main character
        PlayerController player = FindObjectOfType<PlayerController>();
        if (player != null)
        {
            target = player.transform;
        }
        else
        {
            // Fallback to any character
            CharacterController character =
FindObjectOfType<CharacterController>();
            if (character != null)
            {
                target = character.transform;
            }
```

```
    }
}

public void SetTarget(Transform newTarget)
{
    target = newTarget;
}

public void SetMode(MovementMode mode)
{
    currentMode = mode;
}

public void EnableRespectfulMode(bool enabled)
{
    respectPrayerTimes = enabled;
    maintainRespectfulAngles = enabled;
}

float GetPrayerTimeWeight()
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0f;
}

bool IsPrayerTime()
{
    return GetPrayerTimeWeight() > 0.7f;
```

```
    }

    bool IsReligiousContext()
    {
        return GetPrayerTimeWeight() > 0.5f;
    }

    int GetCurrentIslandID()
    {
        // Get current island ID from game state
        return 0; // Placeholder
    }

    public override void OnStateChanged(GameState newState, GameState
oldState)
    {
        base.OnStateChanged(newState, oldState);

        switch (newState)
        {
            case GameState.PrayerTime:
                SetMode(MovementMode.Prayer);
                break;

            case GameState.CulturalEvent:
                SetMode(MovementMode.Respectful);
                break;

            case GameState.Playing:
                SetMode(MovementMode.Follow);
                break;
```

```
        }
    }

    public CameraSnapshot GetCameraSnapshot()
    {
        return new CameraSnapshot
        {
            current_mode = GetCurrentMode().ToString(),
            target_locked = target != null,
            zoom_level = currentZoom,
            is_prayer_time = isPrayerTime,
            respectful_mode = respectPrayerTimes,
            cultural_context = IsReligiousContext()
        };
    }

    [System.Serializable]
    public class CameraSnapshot
    {
        public string current_mode;
        public bool target_locked;
        public float zoom_level;
        public bool is_prayer_time;
        public bool respectful_mode;
        public bool cultural_context;
    }
}
```

50. AudioManager.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;

[BurstCompile]
public class AudioManager : GameSystem
{
    [BurstCompile]
    struct AudioOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> audioSourceDistances;
        [ReadOnly] public NativeArray<bool> isActiveSound;
        [ReadOnly] public NativeArray<int> soundPriorities;
        [ReadOnly] public NativeArray<float> soundVolumes;
        [WriteOnly] public NativeArray<bool> shouldPlaySound;
        [WriteOnly] public NativeArray<float> optimizedVolumes;
        [WriteOnly] public NativeArray<int> audioOptimizationPriority;

        public float maxDistance;
        public float maxConcurrentSounds;
        public float performanceThreshold;
        public float3 listenerPosition;

        public void Execute(int index)
        {
            float distance = audioSourceDistances[index];
            bool active = isActiveSound[index];
            int priority = soundPriorities[index];
```

```
            float volume = soundVolumes[index];

            bool shouldPlay = ShouldPlaySound(distance, active, priority);
            float optimizedVolume = OptimizeVolume(distance, volume);
            int optimizationPriority = CalculateAudioPriority(distance, priority);

            shouldPlaySound[index] = shouldPlay;
            optimizedVolumes[index] = optimizedVolume;
            audioOptimizationPriority[index] = optimizationPriority;
        }

        [BurstCompile]
        bool ShouldPlaySound(float distance, bool active, int priority)
        {
            bool distanceOk = distance < maxDistance;
            bool priorityOk = priority > 5 || active;
            bool performanceOk = priority > 3; // High priority sounds always play

            return distanceOk && priorityOk && performanceOk;
        }

        [BurstCompile]
        float OptimizeVolume(float distance, float originalVolume)
        {
            float distanceAttenuation = math.saturate(1.0f - (distance /
maxDistance));
            return originalVolume * distanceAttenuation;
        }

        [BurstCompile]
        int CalculateAudioPriority(float distance, int priority)
```

```
    {
        int distancePriority = (int)((maxDistance - distance) / maxDistance * 10);
        return math.max(priority, distancePriority);
    }
}

[BurstCompile]
struct MaldivianCulturalAudio : IJob
{
    public NativeArray<bool> prayerTimeQuiet;
    public NativeArray<bool> traditionalMusicRespect;
    public NativeArray<bool> religiousSoundAppropriate;
    public NativeArray<bool> communityEventAudio;
    public NativeArray<bool> environmentalSoundAwareness;

    public NativeArray<float> culturalSensitivityScores;
    public NativeArray<bool> isAudioRespectful;
    public NativeArray<int> appropriateAudioModes;

    public float currentPrayerTimeWeight;
    public bool isReligiousContext;
    public bool isCommunityEvent;
    public int currentIslandID;
    public float currentTime;

    public void Execute()
    {
        ValidatePrayerTimeQuiet();
        ValidateTraditionalMusicRespect();
        ValidateReligiousSoundAppropriateness();
        CalculateAudioRespectfulness();
```

```
    DetermineAppropriateAudioMode();
}

[BurstCompile]
void ValidatePrayerTimeQuiet()
{
    for (int i = 0; i < prayerTimeQuiet.Length; i++)
    {
        bool shouldBeQuiet = currentPrayerTimeWeight > 0.7f;
        prayerTimeQuiet[i] = shouldBeQuiet;

        if (shouldBeQuiet)
        {
            culturalSensitivityScores[i] = 0.95f;
        }
    }
}

[BurstCompile]
void ValidateTraditionalMusicRespect()
{
    for (int i = 0; i < traditionalMusicRespect.Length; i++)
    {
        bool shouldRespect = isReligiousContext || isCommunityEvent;
        traditionalMusicRespect[i] = shouldRespect;

        if (shouldRespect)
        {
            isAudioRespectful[i] = true;
        }
    }
```

```
    }

    [BurstCompile]
    void ValidateReligiousSoundAppropriateness()
    {
        for (int i = 0; i < religiousSoundAppropriate.Length; i++)
        {
            bool isAppropriate = !isReligiousContext && currentPrayerTimeWeight
< 0.5f;
            religiousSoundAppropriate[i] = isAppropriate;

            if (isAppropriate)
            {
                culturalSensitivityScores[prayerTimeQuiet.Length + i] = 0.90f;
            }
        }
    }

    [BurstCompile]
    void CalculateAudioRespectfulness()
    {
        float totalScore = 0.0f;
        int totalChecks = 0;

        for (int i = 0; i < prayerTimeQuiet.Length; i++)
        {
            if (prayerTimeQuiet[i]) totalScore += 1.0f;
            totalChecks++;
        }

        for (int i = 0; i < traditionalMusicRespect.Length; i++)
```

```
        {
            if (traditionalMusicRespect[i]) totalScore += 1.0f;
            totalChecks++;
        }

        float averageScore = totalChecks > 0 ? totalScore / totalChecks : 0.0f;

        for (int i = 0; i < isAudioRespectful.Length; i++)
        {
            isAudioRespectful[i] = averageScore > 0.8f;
        }
    }

    [BurstCompile]
    void DetermineAppropriateAudioMode()
    {
        int mode = 1; // Normal audio

        if (currentPrayerTimeWeight > 0.7f) mode = 2; // Quiet mode
        if (isReligiousContext) mode = 3; // Respectful mode
        if (isCommunityEvent) mode = 4; // Community mode

        for (int i = 0; i < appropriateAudioModes.Length; i++)
        {
            appropriateAudioModes[i] = mode;
        }
    }
}

public static AudioManager Instance { get; private set; }
```

```
[Header("Audio Settings")]
public int maxConcurrentSounds = 32;
public float masterVolume = 1f;
public float musicVolume = 0.7f;
public float sfxVolume = 0.8f;
public float ambientVolume = 0.5f;
public bool enable3DAudio = true;
public bool enableDucking = true;

[Header("Cultural Audio")]
public bool respectPrayerTimes = true;
public bool enableTraditionalSounds = true;
public bool maintainCulturalAuthenticity = true;
public float prayerTimeVolumeMultiplier = 0.3f;

[Header("Mobile Optimization")]
public bool optimizeForMobile = true;
public int maxMobileSounds = 16;
public bool reduceAudioQuality = false;
public bool disableReverb = true;

private AudioSource musicSource;
private AudioSource ambientSource;
private List<AudioSource> sfxSources;
private Dictionary<string, AudioClip> soundLibrary;
private Dictionary<string, AudioSource> loopingsSounds;
private Transform listenerTransform;
private bool isPrayerTime;
private AudioMode currentMode;

public enum AudioMode
```

```
{
    Normal = 0,
    Quiet = 1,
    Respectful = 2,
    Prayer = 3,
    Community = 4
}

public enum SoundType
{
    Music = 0,
    SFX = 1,
    Ambient = 2,
    UI = 3,
    Cultural = 4,
    Religious = 5,
    Traditional = 6
}

void Awake()
{
    if (Instance == null)
    {
        Instance = this;
        DontDestroyOnLoad(gameObject);
    }
    else
    {
        Destroy(gameObject);
    }
}
```

```
    public override void Initialize(MainGameManager manager)
    {
        base.Initialize(manager);
        InitializeAudioManager();
    }

    void InitializeAudioManager()
    {
        // Create audio sources
        CreateAudioSources();

        // Initialize collections
        soundLibrary = new Dictionary<string, AudioClip>();
        loopingsSounds = new Dictionary<string, AudioSource>();
        sfxSources = new List<AudioSource>();

        // Initialize audio optimization
        int soundCount = 50;
        var distances = new NativeArray<float>(soundCount, Allocator.TempJob);
        var activeStates = new NativeArray<bool>(soundCount,
Allocator.TempJob);
        var priorities = new NativeArray<int>(soundCount, Allocator.TempJob);
        var volumes = new NativeArray<float>(soundCount, Allocator.TempJob);
        var shouldPlay = new NativeArray<bool>(soundCount, Allocator.TempJob);
        var optimizedVols = new NativeArray<float>(soundCount,
Allocator.TempJob);
        var audioPriorities = new NativeArray<int>(soundCount,
Allocator.TempJob);

        // Initialize with sample audio data
```

```
        for (int i = 0; i < soundCount; i++)
        {
            distances[i] = UnityEngine.Random.Range(1f, 100f);
            activeStates[i] = UnityEngine.Random.Range(0, 2) == 1;
            priorities[i] = UnityEngine.Random.Range(1, 10);
            volumes[i] = UnityEngine.Random.Range(0.1f, 1f);
        }

        var optimizationJob = new AudioOptimization
        {
            audioSourceDistances = distances,
            isActiveSound = activeStates,
            soundPriorities = priorities,
            soundVolumes = volumes,
            shouldPlaySound = shouldPlay,
            optimizedVolumes = optimizedVols,
            audioOptimizationPriority = audioPriorities,
            maxDistance = 100f,
            maxConcurrentSounds = optimizeForMobile ? maxMobileSounds :
maxConcurrentSounds,
            performanceThreshold = 0.016f,
            listenerPosition = listenerTransform ? listenerTransform.position :
Vector3.zero
        };

        // Initialize cultural audio validation
        var prayerQuiet = new NativeArray<bool>(6, Allocator.TempJob);
        var traditionalRespect = new NativeArray<bool>(8, Allocator.TempJob);
        var religiousAppropriate = new NativeArray<bool>(4, Allocator.TempJob);
        var communityAudio = new NativeArray<bool>(5, Allocator.TempJob);
```

```
        var environmentalAwareness = new NativeArray<bool>(7,
Allocator.TempJob);
        var sensitivityScores = new NativeArray<float>(50, Allocator.TempJob);
        var audioRespectful = new NativeArray<bool>(50, Allocator.TempJob);
        var audioModes = new NativeArray<int>(50, Allocator.TempJob);

        var culturalJob = new MaldivianCulturalAudio
        {
            prayerTimeQuiet = prayerQuiet,
            traditionalMusicRespect = traditionalRespect,
            religiousSoundAppropriate = religiousAppropriate,
            communityEventAudio = communityAudio,
            environmentalSoundAwareness = environmentalAwareness,
            culturalSensitivityScores = sensitivityScores,
            isAudioRespectful = audioRespectful,
            appropriateAudioModes = audioModes,
            currentPrayerTimeWeight = GetPrayerTimeWeight(),
            isReligiousContext = IsReligiousContext(),
            isCommunityEvent = IsCommunityEvent(),
            currentIslandID = GetCurrentIslandID(),
            currentTime = Time.time
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(soundCount, 8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessOptimizationResults(shouldPlay, optimizedVols, audioPriorities);
        ProcessCulturalValidation(prayerQuiet, traditionalRespect,
sensitivityScores, audioModes);
```

```
    // Cleanup
    distances.Dispose();
    activeStates.Dispose();
    priorities.Dispose();
    volumes.Dispose();
    shouldPlay.Dispose();
    optimizedVols.Dispose();
    audioPriorities.Dispose();
    prayerQuiet.Dispose();
    traditionalRespect.Dispose();
    religiousAppropriate.Dispose();
    communityAudio.Dispose();
    environmentalAwareness.Dispose();
    sensitivityScores.Dispose();
    audioRespectful.Dispose();
    audioModes.Dispose();

    StartAudioMonitoring();
}

void CreateAudioSources()
{
    // Create main audio sources
    musicSource = gameObject.AddComponent<AudioSource>();
    musicSource.loop = true;
    musicSource.playOnAwake = false;
    musicSource.priority = 0; // Highest priority

    ambientSource = gameObject.AddComponent<AudioSource>();
    ambientSource.loop = true;
```

```
        ambientSource.playOnAwake = false;
        ambientSource.priority = 1;

        // Create listener if needed
        AudioListener listener = FindObjectOfType<AudioListener>();
        if (listener == null)
        {
            GameObject listenerObject = new GameObject("AudioListener");
            listener = listenerObject.AddComponent<AudioListener>();
        }

        listenerTransform = listener.transform;
    }

    void ProcessOptimizationResults(NativeArray<bool> shouldPlay,
NativeArray<float> volumes, NativeArray<int> priorities)
    {
        int playingCount = 0;
        for (int i = 0; i < shouldPlay.Length; i++)
        {
            if (shouldPlay[i]) playingCount++;
        }

        Debug.Log($"Audio optimization: {playingCount} sounds should play");
    }

    void ProcessCulturalValidation(NativeArray<bool> prayer, NativeArray<bool>
traditional, NativeArray<float> sensitivity, NativeArray<int> modes)
    {
        int quietCount = 0;
        for (int i = 0; i < prayer.Length; i++)
```

```
        {
            if (prayer[i]) quietCount++;
        }

        int respectfulCount = 0;
        for (int i = 0; i < traditional.Length; i++)
        {
            if (traditional[i]) respectfulCount++;
        }

        Debug.Log($"Cultural validation: {quietCount} prayer quiet modes,
{respectfulCount} traditional respects");
    }

    void StartAudioMonitoring()
    {
        InvokeRepeating("UpdateAudioOptimization", 0.5f, 0.2f);
        InvokeRepeating("CheckCulturalContext", 1f, 5f);
        InvokeRepeating("ValidateAudioRespect", 2f, 10f);
    }

    void Update()
    {
        if (!gameManager) return;

        UpdateAudioVolumes();
        CheckCulturalContext();
    }

    void UpdateAudioOptimization()
    {
```

```
    // Optimize active audio sources
    if (optimizeForMobile)
    {
        OptimizeActiveSounds();
    }
}

void OptimizeActiveSounds()
{
    // Limit concurrent sounds on mobile
    int activeCount = 0;
    foreach (var source in sfxSources)
    {
        if (source.isPlaying) activeCount++;
    }

    if (activeCount > maxMobileSounds)
    {
        // Reduce least important sounds
        ReduceSoundPriority();
    }
}

void ReduceSoundPriority()
{
    // Find and stop least important sounds
    AudioSource leastImportant = null;
    int lowestPriority = int.MaxValue;

    foreach (var source in sfxSources)
    {
```

```
        if (source.isPlaying && source.priority < lowestPriority)
        {
            lowestPriority = source.priority;
            leastImportant = source;
        }
    }

    if (leastImportant != null)
    {
        leastImportant.Stop();
    }
}

void CheckCulturalContext()
{
    // Update cultural context
    AudioMode newMode = GetAudioMode();

    if (newMode != currentMode)
    {
        SetAudioMode(newMode);
    }
}

AudioMode GetAudioMode()
{
    if (respectPrayerTimes && IsPrayerTime())
    {
        return AudioMode.Prayer;
    }
```

```
    if (IsReligiousContext())
    {
        return AudioMode.Respectful;
    }

    if (IsCommunityEvent())
    {
        return AudioMode.Community;
    }

    return AudioMode.Normal;
}

void SetAudioMode(AudioMode mode)
{
    currentMode = mode;

    switch (mode)
    {
        case AudioMode.Prayer:
            SetPrayerMode();
            break;

        case AudioMode.Respectful:
            SetRespectfulMode();
            break;

        case AudioMode.Community:
            SetCommunityMode();
            break;
```

```
        default:
            SetNormalMode();
            break;
    }
}

void SetPrayerMode()
{
    // Reduce all audio during prayer time
    musicSource.volume = musicVolume * prayerTimeVolumeMultiplier;
    ambientSource.volume = ambientVolume * prayerTimeVolumeMultiplier;

    // Duck SFX sounds
    foreach (var source in sfxSources)
    {
        if (source.isPlaying)
        {
            source.volume *= prayerTimeVolumeMultiplier;
        }
    }
}

void SetRespectfulMode()
{
    // Moderate volume reduction
    musicSource.volume = musicVolume * 0.7f;
    ambientSource.volume = ambientVolume * 0.8f;
}

void SetCommunityMode()
{
```

```
        // Allow community sounds, reduce music
        musicSource.volume = musicVolume * 0.5f;
        ambientSource.volume = ambientVolume * 1.2f; // Increase ambient
    }

    void SetNormalMode()
    {
        // Normal volumes
        musicSource.volume = musicVolume;
        ambientSource.volume = ambientVolume;
    }

    void UpdateAudioVolumes()
    {
        // Update master volume
        AudioListener.volume = masterVolume;
    }

    public void PlaySound(string soundName, SoundType type = SoundType.SFX,
Vector3 position = default)
    {
        if (!CanPlaySound(type)) return;

        AudioClip clip = GetSoundClip(soundName);
        if (clip == null) return;

        AudioSource source = GetAvailableSource();
        if (source == null) return;

        source.clip = clip;
        source.volume = GetVolumeForType(type);
```

```
    source.priority = GetPriorityForType(type);

    if (position != Vector3.zero && enable3DAudio)
    {
        source.transform.position = position;
        source.spatialBlend = 1f;
    }
    else
    {
        source.spatialBlend = 0f;
    }

    source.Play();
}

public void PlayMusic(string musicName, bool loop = true)
{
    if (!CanPlaySound(SoundType.Music)) return;

    AudioClip clip = GetSoundClip(musicName);
    if (clip == null) return;

    musicSource.clip = clip;
    musicSource.loop = loop;
    musicSource.volume = GetVolumeForType(SoundType.Music);
    musicSource.Play();
}

public void PlayAmbient(string ambientName, bool loop = true)
{
    if (!CanPlaySound(SoundType.Ambient)) return;
```

```
    AudioClip clip = GetSoundClip(ambientName);
    if (clip == null) return;

    ambientSource.clip = clip;
    ambientSource.loop = loop;
    ambientSource.volume = GetVolumeForType(SoundType.Ambient);
    ambientSource.Play();
}

bool CanPlaySound(SoundType type)
{
    switch (currentMode)
    {
        case AudioMode.Prayer:
            return type == SoundType.Religious || type == SoundType.Traditional;

        case AudioMode.Respectful:
            return type != SoundType.Music; // No music during respectful mode

        default:
            return true;
    }
}

AudioClip GetSoundClip(string soundName)
{
    if (soundLibrary.ContainsKey(soundName))
    {
        return soundLibrary[soundName];
    }
```

```
    // Try to load from Resources
    AudioClip clip = Resources.Load<AudioClip>($"Sounds/{soundName}");
    if (clip != null)
    {
        soundLibrary[soundName] = clip;
    }

    return clip;
}

AudioSource GetAvailableSource()
{
    // Find or create available SFX source
    foreach (var source in sfxSources)
    {
        if (!source.isPlaying)
        {
            return source;
        }
    }

    // Create new source if needed
    if (sfxSources.Count < maxConcurrentSounds)
    {
        AudioSource newSource = gameObject.AddComponent<AudioSource>();
        sfxSources.Add(newSource);
        return newSource;
    }

    return null;
```

```
}

float GetVolumeForType(SoundType type)
{
    return type switch
    {
        SoundType.Music => musicVolume,
        SoundType.SFX => sfxVolume,
        SoundType.Ambient => ambientVolume,
        SoundType.UI => sfxVolume,
        SoundType.Cultural => sfxVolume * 1.2f,
        SoundType.Religious => sfxVolume * 0.8f,
        SoundType.Traditional => sfxVolume * 1.1f,
        _ => sfxVolume
    };
}

int GetPriorityForType(SoundType type)
{
    return type switch
    {
        SoundType.Music => 0,
        SoundType.UI => 10,
        SoundType.Religious => 20,
        SoundType.Cultural => 30,
        SoundType.Traditional => 40,
        SoundType.SFX => 50,
        SoundType.Ambient => 60,
        _ => 50
    };
}
```

```
    public void StopSound(string soundName)
    {
        foreach (var source in sfxSources)
        {
            if (source.isPlaying && source.clip != null && source.clip.name ==
soundName)
            {
                source.Stop();
            }
        }
    }

    public void StopMusic()
    {
        musicSource.Stop();
    }

    public void StopAmbient()
    {
        ambientSource.Stop();
    }

    public void SetMasterVolume(float volume)
    {
        masterVolume = Mathf.Clamp01(volume);
    }

    public void SetMusicVolume(float volume)
    {
        musicVolume = Mathf.Clamp01(volume);
```

```
    musicSource.volume = musicVolume;
}

public void SetSFXVolume(float volume)
{
    sfxVolume = Mathf.Clamp01(volume);
}

public void SetAmbientVolume(float volume)
{
    ambientVolume = Mathf.Clamp01(volume);
    ambientSource.volume = ambientVolume;
}

public void EnableRespectfulMode(bool enabled)
{
    respectPrayerTimes = enabled;
    maintainCulturalAuthenticity = enabled;
}

float GetPrayerTimeWeight()
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0f;
}

bool IsPrayerTime()
{
```

```
    return GetPrayerTimeWeight() > 0.7f;
}

bool IsReligiousContext()
{
    return GetPrayerTimeWeight() > 0.5f;
}

bool IsCommunityEvent()
{
    // Simplified check
    return Time.time % 86400 > 43200 && Time.time % 86400 < 46800;
}

void ValidateAudioRespect()
{
    // Validate that audio playback is culturally respectful
    if (maintainCulturalAuthenticity)
    {
        bool isRespectfulTime = !IsPrayerTime();

        if (!isRespectfulTime)
        {
            // Ensure only appropriate sounds are playing
            ValidatePrayerTimeAudio();
        }
    }
}

void ValidatePrayerTimeAudio()
{
```

```
    // Stop inappropriate sounds during prayer time
    foreach (var source in sfxSources)
    {
        if (source.isPlaying && source.priority > 30) // Non-essential sounds
        {
            source.Stop();
        }
    }
}

public AudioSnapshot GetAudioSnapshot()
{
    int activeSounds = 0;
    foreach (var source in sfxSources)
    {
        if (source.isPlaying) activeSounds++;
    }

    return new AudioSnapshot
    {
        master_volume = masterVolume,
        music_playing = musicSource.isPlaying,
        ambient_playing = ambientSource.isPlaying,
        active_sounds = activeSounds,
        current_mode = currentMode.ToString(),
        prayer_time_active = IsPrayerTime(),
        cultural_respect = maintainCulturalAuthenticity
    };
}

[System.Serializable]
```

```
    public class AudioSnapshot
    {
        public float master_volume;
        public bool music_playing;
        public bool ambient_playing;
        public int active_sounds;
        public string current_mode;
        public bool prayer_time_active;
        public bool cultural_respect;
    }
}
```

51. UIManager.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using UnityEngine.UI;
using System.Collections.Generic;

[BurstCompile]
public class UIManager : GameSystem
{
    [BurstCompile]
    struct UIOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float2> uiElementPositions;
        [ReadOnly] public NativeArray<float2> uiElementSizes;
        [ReadOnly] public NativeArray<int> elementTypes;
```

```
[ReadOnly] public NativeArray<bool> isActive;
[WriteOnly] public NativeArray<bool> shouldUpdateElement;
[WriteOnly] public NativeArray<float> optimizationScores;
[WriteOnly] public NativeArray<int> renderPriorities;

public float2 screenResolution;
public float performanceThreshold;
public int maxActiveElements;
public float deltaTime;

public void Execute(int index)
{
    float2 position = uiElementPositions[index];
    float2 size = uiElementSizes[index];
    int type = elementTypes[index];
    bool active = isActive[index];

    bool shouldUpdate = ShouldUpdateElement(position, size, type, active);
    float score = CalculateOptimizationScore(position, size, type, active);
    int priority = CalculateRenderPriority(position, size, type);

    shouldUpdateElement[index] = shouldUpdate;
    optimizationScores[index] = score;
    renderPriorities[index] = priority;
}

[BurstCompile]
bool ShouldUpdateElement(float2 position, float2 size, int type, bool active)
{
    if (!active) return false;
```

```
        bool onScreen = IsElementOnScreen(position, size);
        bool importantType = type < 3; // Critical UI elements

        return onScreen || importantType;
    }

    [BurstCompile]
    bool IsElementOnScreen(float2 position, float2 size)
    {
        return position.x + size.x > 0 && position.x < screenResolution.x &&
               position.y + size.y > 0 && position.y < screenResolution.y;
    }

    [BurstCompile]
    float CalculateOptimizationScore(float2 position, float2 size, int type, bool
active)
    {
        if (!active) return 0f;

        float sizeScore = math.saturate((size.x * size.y) / (screenResolution.x *
screenResolution.y * 0.1f));
        float typeScore = (10 - type) / 10f;
        float positionScore = math.saturate(1.0f - (math.length(position -
screenResolution * 0.5f) / math.length(screenResolution * 0.5f)));

        return (sizeScore + typeScore + positionScore) * 0.33f;
    }

    [BurstCompile]
    int CalculateRenderPriority(float2 position, float2 size, int type)
    {
```

```
            int basePriority = type * 10;
            float centerDistance = math.length(position - screenResolution * 0.5f);
            float centerBonus = (1.0f - math.saturate(centerDistance /
math.length(screenResolution * 0.5f))) * 5;

            return basePriority + (int)centerBonus;
        }
    }

    [BurstCompile]
    struct MaldivianCulturalUI : IJob
    {
        public NativeArray<bool> prayerUIDiscretion;
        public NativeArray<bool> culturalContentRespect;
        public NativeArray<bool> traditionalSymbolDisplay;
        public NativeArray<bool> communityNotificationTiming;
        public NativeArray<bool> religiousImageryAppropriate;

        public NativeArray<float> culturalSensitivityScores;
        public NativeArray<bool> isUIDisplayRespectful;
        public NativeArray<int> appropriateUIDisplayModes;

        public float currentPrayerTimeWeight;
        public bool isReligiousContext;
        public bool isCommunityEvent;
        public int currentIslandID;
        public float2 currentScreenPosition;

        public void Execute()
        {
            ValidatePrayerUIDiscretion();
```

```
    ValidateCulturalContentRespect();
    ValidateTraditionalSymbolDisplay();
    CalculateUIDisplayRespectfulness();
    DetermineAppropriateUIDisplayMode();
}

[BurstCompile]
void ValidatePrayerUIDiscretion()
{
    for (int i = 0; i < prayerUIDiscretion.Length; i++)
    {
        bool shouldBeDiscreet = currentPrayerTimeWeight > 0.7f;
        prayerUIDiscretion[i] = shouldBeDiscreet;

        if (shouldBeDiscreet)
        {
            culturalSensitivityScores[i] = 0.95f;
        }
    }
}

[BurstCompile]
void ValidateCulturalContentRespect()
{
    for (int i = 0; i < culturalContentRespect.Length; i++)
    {
        bool shouldRespect = isReligiousContext || isCommunityEvent;
        culturalContentRespect[i] = shouldRespect;

        if (shouldRespect)
        {
```

```
            isUIDisplayRespectful[i] = true;
        }
    }
}

[BurstCompile]
void ValidateTraditionalSymbolDisplay()
{
    for (int i = 0; i < traditionalSymbolDisplay.Length; i++)
    {
        bool isAppropriate = currentIslandID >= 0 && currentIslandID < 41;
        traditionalSymbolDisplay[i] = isAppropriate;

        if (isAppropriate)
        {
            culturalSensitivityScores[prayerUIDiscretion.Length + i] = 0.90f;
        }
    }
}

[BurstCompile]
void CalculateUIDisplayRespectfulness()
{
    float totalScore = 0.0f;
    int totalChecks = 0;

    for (int i = 0; i < prayerUIDiscretion.Length; i++)
    {
        if (prayerUIDiscretion[i]) totalScore += 1.0f;
        totalChecks++;
    }
```

```
        for (int i = 0; i < culturalContentRespect.Length; i++)
        {
            if (culturalContentRespect[i]) totalScore += 1.0f;
            totalChecks++;
        }

        float averageScore = totalChecks > 0 ? totalScore / totalChecks : 0.0f;

        for (int i = 0; i < isUIDisplayRespectful.Length; i++)
        {
            isUIDisplayRespectful[i] = averageScore > 0.8f;
        }
    }

    [BurstCompile]
    void DetermineAppropriateUIDisplayMode()
    {
        int mode = 1; // Normal UI display

        if (currentPrayerTimeWeight > 0.7f) mode = 2; // Discreet mode
        if (isReligiousContext) mode = 3; // Respectful mode
        if (isCommunityEvent) mode = 4; // Community-aware mode

        for (int i = 0; i < appropriateUIDisplayModes.Length; i++)
        {
            appropriateUIDisplayModes[i] = mode;
        }
    }
}
```

```
public static UIManager Instance { get; private set; }

[Header("UI Canvas")]
public Canvas mainCanvas;
public Canvas overlayCanvas;
public Canvas prayerCanvas;
public Canvas culturalCanvas;

[Header("Core UI Panels")]
public GameObject mainMenuPanel;
public GameObject hudPanel;
public GameObject pausePanel;
public GameObject loadingPanel;
public GameObject settingsPanel;

[Header("Cultural UI Elements")]
public GameObject prayerTimeNotification;
public GameObject culturalEventPanel;
public GameObject respectfulModeIndicator;
public GameObject communityNotification;

[Header("UI Settings")]
public bool respectPrayerTimes = true;
public bool enableCulturalUI = true;
public bool maintainRespectfulDisplay = true;
public float uiUpdateInterval = 0.1f;

[Header("Mobile Optimization")]
public bool optimizeForMobile = true;
public bool reduceAnimations = false;
public bool simplifyUIElements = false;
```

```
public int maxActiveUIElements = 50;

private Dictionary<string, GameObject> uiElements;
private Dictionary<string, Text> textElements;
private Dictionary<string, Image> imageElements;
private Dictionary<string, Button> buttonElements;
private Stack<GameObject> panelHistory;
private GameObject currentPanel;
private bool isPrayerTime;
private bool isCulturalEvent;
private UIDisplayMode currentMode;

public enum UIDisplayMode
{
    Normal = 0,
    Discreet = 1,
    Respectful = 2,
    Prayer = 3,
    Community = 4
}

void Awake()
{
    if (Instance == null)
    {
        Instance = this;
    }
    else
    {
        Destroy(gameObject);
    }
```

```
    }

    public override void Initialize(MainGameManager manager)
    {
        base.Initialize(manager);
        InitializeUIManager();
    }

    void InitializeUIManager()
    {
        // Initialize collections
        uiElements = new Dictionary<string, GameObject>();
        textElements = new Dictionary<string, Text>();
        imageElements = new Dictionary<string, Image>();
        buttonElements = new Dictionary<string, Button>();
        panelHistory = new Stack<GameObject>();

        // Initialize UI optimization
        int elementCount = 100;
        var positions = new NativeArray<float2>(elementCount, Allocator.TempJob);
        var sizes = new NativeArray<float2>(elementCount, Allocator.TempJob);
        var types = new NativeArray<int>(elementCount, Allocator.TempJob);
        var active = new NativeArray<bool>(elementCount, Allocator.TempJob);
        var shouldUpdate = new NativeArray<bool>(elementCount,
Allocator.TempJob);
        var scores = new NativeArray<float>(elementCount, Allocator.TempJob);
        var priorities = new NativeArray<int>(elementCount, Allocator.TempJob);

        // Initialize with sample UI data
        for (int i = 0; i < elementCount; i++)
        {
```

```
    positions[i] = UnityEngine.Random.insideUnitCircle * 1000f;
    sizes[i] = UnityEngine.Random.Range(50f, 200f) * Vector2.one;
    types[i] = UnityEngine.Random.Range(0, 10);
    active[i] = UnityEngine.Random.Range(0, 2) == 1;
}

var optimizationJob = new UIOptimization
{
    uiElementPositions = positions,
    uiElementSizes = sizes,
    elementTypes = types,
    isActive = active,
    shouldUpdateElement = shouldUpdate,
    optimizationScores = scores,
    renderPriorities = priorities,
    screenResolution = new float2(Screen.width, Screen.height),
    performanceThreshold = 0.016f,
    maxActiveElements = optimizeForMobile ? maxActiveUIElements : 100,
    deltaTime = Time.deltaTime
};

// Initialize cultural UI validation
var prayerDiscretion = new NativeArray<bool>(8, Allocator.TempJob);
var culturalRespect = new NativeArray<bool>(10, Allocator.TempJob);
var traditionalDisplay = new NativeArray<bool>(12, Allocator.TempJob);
var communityTiming = new NativeArray<bool>(6, Allocator.TempJob);
var religiousImagery = new NativeArray<bool>(4, Allocator.TempJob);
var sensitivityScores = new NativeArray<float>(60, Allocator.TempJob);
uiRespectful = new NativeArray<bool>(60, Allocator.TempJob);
var uiModes = new NativeArray<int>(60, Allocator.TempJob);
```

```
        var culturalJob = new MaldivianCulturalUI
        {
            prayerUIDiscretion = prayerDiscretion,
            culturalContentRespect = culturalRespect,
            traditionalSymbolDisplay = traditionalDisplay,
            communityNotificationTiming = communityTiming,
            religiousImageryAppropriate = religiousImagery,
            culturalSensitivityScores = sensitivityScores,
            isUIDisplayRespectful = uiRespectful,
            appropriateUIDisplayModes = uiModes,
            currentPrayerTimeWeight = GetPrayerTimeWeight(),
            isReligiousContext = IsReligiousContext(),
            isCommunityEvent = IsCommunityEvent(),
            currentIslandID = GetCurrentIslandID(),
            currentScreenPosition = new float2(Input.mousePosition.x,
Input.mousePosition.y)
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(elementCount,
8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessOptimizationResults(shouldUpdate, scores, priorities);
        ProcessCulturalValidation(prayerDiscretion, culturalRespect,
sensitivityScores, uiModes);

        // Cleanup
        positions.Dispose();
        sizes.Dispose();
```

```
        types.Dispose();
        active.Dispose();
        shouldUpdate.Dispose();
        scores.Dispose();
        priorities.Dispose();
        prayerDiscretion.Dispose();
        culturalRespect.Dispose();
        traditionalDisplay.Dispose();
        communityTiming.Dispose();
        religiousImagery.Dispose();
        sensitivityScores.Dispose();
        uiRespectful.Dispose();
        uiModes.Dispose();

        RegisterUIElements();
        StartUIMonitoring();
    }

    void ProcessOptimizationResults(NativeArray<bool> shouldUpdate,
NativeArray<float> scores, NativeArray<int> priorities)
    {
        int updateCount = 0;
        float avgScore = 0f;

        for (int i = 0; i < shouldUpdate.Length; i++)
        {
            if (shouldUpdate[i]) updateCount++;
            avgScore += scores[i];
        }

        avgScore /= shouldUpdate.Length;
```

```
        Debug.Log($"UI optimization: {updateCount} elements should update,
average score: {avgScore:F2}");
    }

    void ProcessCulturalValidation(NativeArray<bool> prayer, NativeArray<bool>
cultural, NativeArray<float> sensitivity, NativeArray<int> modes)
    {
        int discreetCount = 0;
        for (int i = 0; i < prayer.Length; i++)
        {
            if (prayer[i]) discreetCount++;
        }

        int respectfulCount = 0;
        for (int i = 0; i < cultural.Length; i++)
        {
            if (cultural[i]) respectfulCount++;
        }

        Debug.Log($"Cultural validation: {discreetCount} prayer discreet modes,
{respectfulCount} cultural respects");
    }

    void StartUIMonitoring()
    {
        InvokeRepeating("UpdateUIElements", 0f, uiUpdateInterval);
        InvokeRepeating("CheckCulturalContext", 1f, 5f);
        InvokeRepeating("ValidateUIDisplay", 2f, 10f);
    }
```

```
void Update()
{
    if (!gameManager) return;

    HandleInput();
    UpdateUIVisibility();
    CheckCulturalContext();
}

void HandleInput()
{
    // Handle UI input from InputManager
    if (gameManager.GetSystem<InputManager>() != null)
    {
        var inputManager = gameManager.GetSystem<InputManager>();

        if (inputManager.IsUIInputActive())
        {
            HandleUIInput(inputManager.GetUIInput());
        }
    }
}

void HandleUIInput(Vector2 input)
{
    // Handle UI navigation, button clicks, etc.
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        TogglePauseMenu();
    }
}
```

```
void UpdateUIElements()
{
    // Update UI elements based on game state
    UpdateHUD();
    UpdateCulturalUI();
    ValidateUIDisplay();
}

void UpdateUIVisibility()
{
    // Manage UI visibility based on cultural context
    UIDisplayMode mode = GetDisplayMode();

    if (mode != currentMode)
    {
        SetDisplayMode(mode);
    }
}

void UpdateHUD()
{
    // Update heads-up display elements
    if (hudPanel != null && hudPanel.activeInHierarchy)
    {
        UpdatePlayerInfo();
        UpdateMiniMap();
        UpdateObjectiveTracker();
    }
}
```

```
void UpdatePlayerInfo()
{
    // Update player health, money, etc.
    var player = gameManager.GetSystem<PlayerController>();
    if (player != null)
    {
        SetText("PlayerPosition", player.transform.position.ToString("F1"));
    }
}

void UpdateMiniMap()
{
    // Update minimap display
    SetText("CurrentIsland", $"Island {GetCurrentIslandID()}");
}

void UpdateObjectiveTracker()
{
    // Update current objectives
    SetText("CurrentObjective", "Explore the Maldivian islands");
}

void UpdateCulturalUI()
{
    // Update cultural UI elements
    if (isPrayerTime && prayerTimeNotification != null)
    {
        ShowPrayerNotification();
    }

    if (isCulturalEvent && culturalEventPanel != null)
```

```
    {
        ShowCulturalEventPanel();
    }
}

void CheckCulturalContext()
{
    // Update cultural context
    bool newPrayerTime = IsPrayerTime();
    bool newCulturalEvent = IsCulturalEvent();

    if (newPrayerTime != isPrayerTime)
    {
        isPrayerTime = newPrayerTime;
        OnPrayerTimeChanged(isPrayerTime);
    }

    if (newCulturalEvent != isCulturalEvent)
    {
        isCulturalEvent = newCulturalEvent;
        OnCulturalEventChanged(isCulturalEvent);
    }
}

void OnPrayerTimeChanged(bool isPrayer)
{
    if (respectPrayerTimes)
    {
        if (isPrayer)
        {
            ShowPrayerNotification();
```

```
            EnterDiscreetMode();
        }
        else
        {
            HidePrayerNotification();
            ExitDiscreetMode();
        }
    }
}

void OnCulturalEventChanged(bool isEvent)
{
    if (enableCulturalUI)
    {
        if (isEvent)
        {
            ShowCulturalEventPanel();
        }
        else
        {
            HideCulturalEventPanel();
        }
    }
}

void ShowPrayerNotification()
{
    if (prayerTimeNotification != null)
    {
        prayerTimeNotification.SetActive(true);
        SetText("PrayerNotificationText", "Prayer time - Observing respectfully");
```

```
    }
}

void HidePrayerNotification()
{
    if (prayerTimeNotification != null)
    {
        prayerTimeNotification.SetActive(false);
    }
}

void ShowCulturalEventPanel()
{
    if (culturalEventPanel != null)
    {
        culturalEventPanel.SetActive(true);
    }
}

void HideCulturalEventPanel()
{
    if (culturalEventPanel != null)
    {
        culturalEventPanel.SetActive(false);
    }
}

public void ShowMainMenu()
{
    SwitchPanel(mainMenuPanel);
}
```

```
public void ShowHUD()
{
    SwitchPanel(hudPanel);
}

public void ShowPauseMenu()
{
    SwitchPanel(pausePanel);
}

public void ShowLoadingScreen()
{
    if (loadingPanel != null)
    {
        loadingPanel.SetActive(true);
    }
}

public void HideLoadingScreen()
{
    if (loadingPanel != null)
    {
        loadingPanel.SetActive(false);
    }
}

public void UpdateLoadingProgress(float progress)
{
    if (loadingPanel != null)
    {
```

```
        SetText("LoadingProgress", $"{(progress * 100):F0}%");

        var progressBar = GetUIElement<Image>("LoadingProgressBar");
        if (progressBar != null)
        {
            progressBar.fillAmount = progress;
        }
    }
}

void SwitchPanel(GameObject newPanel)
{
    if (currentPanel != null)
    {
        currentPanel.SetActive(false);
        panelHistory.Push(currentPanel);
    }

    currentPanel = newPanel;
    if (currentPanel != null)
    {
        currentPanel.SetActive(true);
    }
}

public void GoBack()
{
    if (panelHistory.Count > 0)
    {
        GameObject previousPanel = panelHistory.Pop();
        SwitchPanel(previousPanel);
```

```
    }
}

void EnterDiscreetMode()
{
    // Reduce UI visibility during prayer time
    if (maintainRespectfulDisplay)
    {
        SetCanvasAlpha(mainCanvas, 0.7f);
        HideNonEssentialUI();
    }
}

void ExitDiscreetMode()
{
    // Restore normal UI visibility
    SetCanvasAlpha(mainCanvas, 1f);
    ShowEssentialUI();
}

void SetCanvasAlpha(Canvas canvas, float alpha)
{
    var canvasGroup = canvas.GetComponent<CanvasGroup>();
    if (canvasGroup == null)
    {
        canvasGroup = canvas.gameObject.AddComponent<CanvasGroup>();
    }
    canvasGroup.alpha = alpha;
}

void HideNonEssentialUI()
```

```
    {
        // Hide non-essential UI elements during prayer time
        var nonEssentialElements =
GameObject.FindGameObjectsWithTag("NonEssentialUI");
        foreach (var element in nonEssentialElements)
        {
            element.SetActive(false);
        }
    }

    void ShowEssentialUI()
    {
        // Show all UI elements
        var allUIElements = GameObject.FindGameObjectsWithTag("UI");
        foreach (var element in allUIElements)
        {
            element.SetActive(true);
        }
    }

    UIDisplayMode GetDisplayMode()
    {
        if (respectPrayerTimes && IsPrayerTime())
        {
            return UIDisplayMode.Prayer;
        }

        if (IsReligiousContext())
        {
            return UIDisplayMode.Respectful;
        }
```

```
    if (IsCommunityEvent())
    {
        return UIDisplayMode.Community;
    }

    return UIDisplayMode.Normal;
}

void SetDisplayMode(UIDisplayMode mode)
{
    currentMode = mode;

    switch (mode)
    {
        case UIDisplayMode.Prayer:
            SetPrayerMode();
            break;

        case UIDisplayMode.Respectful:
            SetRespectfulMode();
            break;

        case UIDisplayMode.Community:
            SetCommunityMode();
            break;

        default:
            SetNormalMode();
            break;
    }
```

```
}

void SetPrayerMode()
{
    // Discreet UI during prayer time
    if (prayerCanvas != null)
    {
        prayerCanvas.gameObject.SetActive(true);
    }

    // Reduce brightness of main UI
    SetCanvasAlpha(mainCanvas, 0.6f);
}

void SetRespectfulMode()
{
    // Respectful UI display
    SetCanvasAlpha(mainCanvas, 0.8f);
}

void SetCommunityMode()
{
    // Community-focused UI
    if (culturalCanvas != null)
    {
        culturalCanvas.gameObject.SetActive(true);
    }
}

void SetNormalMode()
{
```

```
    // Normal UI display
    SetCanvasAlpha(mainCanvas, 1f);

    if (prayerCanvas != null)
    {
        prayerCanvas.gameObject.SetActive(false);
    }

    if (culturalCanvas != null)
    {
        culturalCanvas.gameObject.SetActive(false);
    }
}

void ValidateUIDisplay()
{
    // Validate that UI display is culturally respectful
    if (maintainRespectfulDisplay)
    {
        bool isRespectfulTime = !IsPrayerTime();

        if (!isRespectfulTime && currentMode != UIDisplayMode.Prayer)
        {
            SetDisplayMode(UIDisplayMode.Prayer);
        }
    }
}

void RegisterUIElements()
{
    // Register all UI elements for easy access
```

```
    RegisterUIElement("MainMenuPanel", mainMenuPanel);
    RegisterUIElement("HUDPanel", hudPanel);
    RegisterUIElement("PausePanel", pausePanel);
    RegisterUIElement("LoadingPanel", loadingPanel);
    RegisterUIElement("SettingsPanel", settingsPanel);

    RegisterUIElement("PrayerNotification", prayerTimeNotification);
    RegisterUIElement("CulturalEventPanel", culturalEventPanel);
    RegisterUIElement("RespectfulModeIndicator", respectfulModeIndicator);
}

void RegisterUIElement(string name, GameObject element)
{
    if (element != null)
    {
        uiElements[name] = element;

        // Find text components
        var texts = element.GetComponentsInChildren<Text>();
        foreach (var text in texts)
        {
            textElements[text.name] = text;
        }

        // Find image components
        var images = element.GetComponentsInChildren<Image>();
        foreach (var image in images)
        {
            imageElements[image.name] = image;
        }
```

```
        // Find button components
        var buttons = element.GetComponentsInChildren<Button>();
        foreach (var button in buttons)
        {
            buttonElements[button.name] = button;
        }
    }
}

public GameObject GetUIElement(string name)
{
    return uiElements.ContainsKey(name) ? uiElements[name] : null;
}

public Text GetTextElement(string name)
{
    return textElements.ContainsKey(name) ? textElements[name] : null;
}

public Image GetUIElement<Image>(string name)
{
    return imageElements.ContainsKey(name) ? imageElements[name] : null;
}

public Button GetButtonElement(string name)
{
    return buttonElements.ContainsKey(name) ? buttonElements[name] : null;
}

public void SetText(string elementName, string text)
{
```

```
    Text textElement = GetTextElement(elementName);
    if (textElement != null)
    {
        textElement.text = text;
    }
}

public void SetImageFill(string elementName, float fillAmount)
{
    Image imageElement = GetUIElement<Image>(elementName);
    if (imageElement != null && imageElement.type == Image.Type.Filled)
    {
        imageElement.fillAmount = fillAmount;
    }
}

public void TogglePauseMenu()
{
    if (pausePanel != null)
    {
        if (pausePanel.activeInHierarchy)
        {
            HidePauseMenu();
        }
        else
        {
            ShowPauseMenu();
        }
    }
}
```

```
void ShowPauseMenu()
{
    if (pausePanel != null)
    {
        pausePanel.SetActive(true);
        Time.timeScale = 0f; // Pause game
    }
}

void HidePauseMenu()
{
    if (pausePanel != null)
    {
        pausePanel.SetActive(false);
        Time.timeScale = 1f; // Resume game
    }
}

float GetPrayerTimeWeight()
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0f;
}

bool IsPrayerTime()
{
    return GetPrayerTimeWeight() > 0.7f;
}
```

```
    bool IsReligiousContext()
    {
        return GetPrayerTimeWeight() > 0.5f;
    }

    bool IsCommunityEvent()
    {
        // Simplified check
        return Time.time % 86400 > 43200 && Time.time % 86400 < 46800;
    }

    int GetCurrentIslandID()
    {
        // Get current island ID from game state
        return 0; // Placeholder
    }

    public override void OnStateChanged(GameState newState, GameState
oldState)
    {
        base.OnStateChanged(newState, oldState);

        switch (newState)
        {
            case GameState.MainMenu:
                ShowMainMenu();
                break;

            case GameState.Playing:
                ShowHUD();
```

```
            break;

        case GameState.Paused:
            ShowPauseMenu();
            break;

        case GameState.PrayerTime:
            ShowPrayerNotification();
            break;

        case GameState.CulturalEvent:
            ShowCulturalEventPanel();
            break;
    }
}

public UISnapshot GetUISnapshot()
{
    return new UISnapshot
    {
        current_panel = currentPanel ? currentPanel.name : "None",
        ui_elements_active = uiElements.Count,
        text_elements = textElements.Count,
        image_elements = imageElements.Count,
        current_mode = currentMode.ToString(),
        prayer_time_active = IsPrayerTime(),
        cultural_respect = maintainRespectfulDisplay
    };
}

[System.Serializable]
```

```
    public class UISnapshot
    {
        public string current_panel;
        public int ui_elements_active;
        public int text_elements;
        public int image_elements;
        public string current_mode;
        public bool prayer_time_active;
        public bool cultural_respect;
    }
}
```

52. InputManager.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;

[BurstCompile]
public class InputManager : GameSystem
{
    [BurstCompile]
    struct InputOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float2> inputPositions;
        [ReadOnly] public NativeArray<float> inputPressures;
        [ReadOnly] public NativeArray<bool> inputActive;
```

```
    [ReadOnly] public NativeArray<float> inputTimestamps;
    [WriteOnly] public NativeArray<bool> validInputs;
    [WriteOnly] public NativeArray<float> processedPositions;
    [WriteOnly] public NativeArray<int> inputPriorities;

    public float2 screenBounds;
    public float maxInputDelay;
    public int maxConcurrentInputs;
    public float inputThreshold;

    public void Execute(int index)
    {
        float2 position = inputPositions[index];
        float pressure = inputPressures[index];
        bool active = inputActive[index];
        float timestamp = inputTimestamps[index];

        bool valid = ValidateInput(position, pressure, active, timestamp);
        float processedPos = ProcessInputPosition(position);
        int priority = CalculateInputPriority(pressure, active);

        validInputs[index] = valid;
        processedPositions[index] = processedPos;
        inputPriorities[index] = priority;
    }

    [BurstCompile]
    bool ValidateInput(float2 position, float pressure, bool active, float
timestamp)
    {
        if (!active) return false;
```

```
        if (pressure < inputThreshold) return false;

        bool inBounds = position.x >= 0 && position.x <= screenBounds.x &&
                        position.y >= 0 && position.y <= screenBounds.y;

        bool notExpired = (Time.time - timestamp) < maxInputDelay;

        return inBounds && notExpired;
    }

    [BurstCompile]
    float ProcessInputPosition(float2 position)
    {
        return math.length(position);
    }

    [BurstCompile]
    int CalculateInputPriority(float pressure, bool active)
    {
        if (!active) return 0;
        return (int)(pressure * 10f);
    }
}

[BurstCompile]
struct MaldivianCulturalInput : IJob
{
    public NativeArray<bool> prayerInputRestrictions;
    public NativeArray<bool> culturalGestureRespect;
    public NativeArray<bool> traditionalInputMethods;
    public NativeArray<bool> communityInteractionInput;
```

```
public NativeArray<bool> religiousContextSensitivity;

public NativeArray<float> culturalSensitivityScores;
public NativeArray<bool> isInputAppropriate;
public NativeArray<int> appropriateInputModes;

public float2 currentInputPosition;
public float currentPrayerTimeWeight;
public bool isReligiousContext;
public bool isCommunityInteraction;
public int currentIslandID;
public float inputPressure;

public void Execute()
{
    ValidatePrayerInputRestrictions();
    ValidateCulturalGestureRespect();
    ValidateTraditionalInputMethods();
    CalculateInputAppropriateness();
    DetermineAppropriateInputMode();
}

[BurstCompile]
void ValidatePrayerInputRestrictions()
{
    for (int i = 0; i < prayerInputRestrictions.Length; i++)
    {
        bool shouldRestrict = currentPrayerTimeWeight > 0.7f;
        prayerInputRestrictions[i] = shouldRestrict;

        if (shouldRestrict)
```

```
        {
            culturalSensitivityScores[i] = 0.95f;
        }
    }
}

[BurstCompile]
void ValidateCulturalGestureRespect()
{
    for (int i = 0; i < culturalGestureRespect.Length; i++)
    {
        bool shouldRespect = isReligiousContext || isCommunityInteraction;
        culturalGestureRespect[i] = shouldRespect;

        if (shouldRespect)
        {
            isInputAppropriate[i] = true;
        }
    }
}

[BurstCompile]
void ValidateTraditionalInputMethods()
{
    for (int i = 0; i < traditionalInputMethods.Length; i++)
    {
        bool isTraditional = currentIslandID >= 0 && currentIslandID < 41;
        traditionalInputMethods[i] = isTraditional;

        if (isTraditional)
        {
```

```
                culturalSensitivityScores[prayerInputRestrictions.Length + i] = 0.90f;
            }
        }
    }

    [BurstCompile]
    void CalculateInputAppropriateness()
    {
        float totalScore = 0.0f;
        int totalChecks = 0;

        for (int i = 0; i < prayerInputRestrictions.Length; i++)
        {
            if (!prayerInputRestrictions[i]) totalScore += 1.0f; // Not restricted =
good
            totalChecks++;
        }

        for (int i = 0; i < culturalGestureRespect.Length; i++)
        {
            if (culturalGestureRespect[i]) totalScore += 1.0f;
            totalChecks++;
        }

        float averageScore = totalChecks > 0 ? totalScore / totalChecks : 0.0f;

        for (int i = 0; i < isInputAppropriate.Length; i++)
        {
            isInputAppropriate[i] = averageScore > 0.7f;
        }
    }
```

```
    [BurstCompile]
    void DetermineAppropriateInputMode()
    {
        int mode = 1; // Normal input

        if (currentPrayerTimeWeight > 0.7f) mode = 2; // Restricted input
        if (isReligiousContext) mode = 3; // Respectful input
        if (isCommunityInteraction) mode = 4; // Community-aware input

        for (int i = 0; i < appropriateInputModes.Length; i++)
        {
            appropriateInputModes[i] = mode;
        }
    }
}

public static InputManager Instance { get; private set; }

[Header("Input Configuration")]
public bool enableGamepad = true;
public bool enableTouch = true;
public bool enableKeyboard = true;
public bool enableMouse = true;
public bool enableGestures = true;

[Header("Cultural Input")]
public bool respectPrayerTimes = true;
public bool enableCulturalGestures = true;
public bool maintainRespectfulInput = true;
public float prayerTimeInputMultiplier = 0.3f;
```

```
[Header("Mobile Optimization")]
public bool optimizeForMobile = true;
public bool reduceInputPrecision = false;
public int inputBufferSize = 10;
public bool disableComplexGestures = false;

private Vector2 movementInput;
private Vector2 cameraInput;
private bool isRunning;
private bool jumpPressed;
private bool interactPressed;
private float zoomInput;
private bool isUIInputActive;
private Vector2 uiInputPosition;
private List<Touch> activeTouches;
private Dictionary<string, KeyCode> keyBindings;
private InputMode currentMode;
private bool isPrayerTime;

public enum InputMode
{
    Normal = 0,
    Respectful = 1,
    Prayer = 2,
    Community = 3,
    Restricted = 4
}

public enum InputType
{
```

```
    Keyboard = 0,
    Mouse = 1,
    Gamepad = 2,
    Touch = 3,
    Gesture = 4
}

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

public override void Initialize(MainGameManager manager)
{
    base.Initialize(manager);
    InitializeInputManager();
}

void InitializeInputManager()
{
    // Initialize collections
    activeTouches = new List<Touch>();
    keyBindings = new Dictionary<string, KeyCode>();
```

```
// Initialize input optimization
int inputCount = 20;
var positions = new NativeArray<float2>(inputCount, Allocator.TempJob);
var pressures = new NativeArray<float>(inputCount, Allocator.TempJob);
var active = new NativeArray<bool>(inputCount, Allocator.TempJob);
var timestamps = new NativeArray<float>(inputCount, Allocator.TempJob);
var valid = new NativeArray<bool>(inputCount, Allocator.TempJob);
var processed = new NativeArray<float>(inputCount, Allocator.TempJob);
var priorities = new NativeArray<int>(inputCount, Allocator.TempJob);

// Initialize with sample input data
for (int i = 0; i < inputCount; i++)
{
    positions[i] = UnityEngine.Random.insideUnitCircle * 1000f;
    pressures[i] = UnityEngine.Random.Range(0f, 1f);
    active[i] = UnityEngine.Random.Range(0, 2) == 1;
    timestamps[i] = Time.time - UnityEngine.Random.Range(0f, 0.1f);
}

var optimizationJob = new InputOptimization
{
    inputPositions = positions,
    inputPressures = pressures,
    inputActive = active,
    inputTimestamps = timestamps,
    validInputs = valid,
    processedPositions = processed,
    inputPriorities = priorities,
    screenBounds = new float2(Screen.width, Screen.height),
    maxInputDelay = 0.1f,
```

```
    maxConcurrentInputs = 5,
    inputThreshold = 0.1f
};

// Initialize cultural input validation
var prayerRestrictions = new NativeArray<bool>(6, Allocator.TempJob);
var gestureRespect = new NativeArray<bool>(8, Allocator.TempJob);
var traditionalMethods = new NativeArray<bool>(10, Allocator.TempJob);
var communityInput = new NativeArray<bool>(5, Allocator.TempJob);
var religiousSensitivity = new NativeArray<bool>(3, Allocator.TempJob);
var sensitivityScores = new NativeArray<float>(50, Allocator.TempJob);
var inputAppropriate = new NativeArray<bool>(50, Allocator.TempJob);
var inputModes = new NativeArray<int>(50, Allocator.TempJob);

var culturalJob = new MaldivianCulturalInput
{
    prayerInputRestrictions = prayerRestrictions,
    culturalGestureRespect = gestureRespect,
    traditionalInputMethods = traditionalMethods,
    communityInteractionInput = communityInput,
    religiousContextSensitivity = religiousSensitivity,
    culturalSensitivityScores = sensitivityScores,
    isInputAppropriate = inputAppropriate,
    appropriateInputModes = inputModes,
    currentInputPosition = Vector2.zero,
    currentPrayerTimeWeight = GetPrayerTimeWeight(),
    isReligiousContext = IsReligiousContext(),
    isCommunityInteraction = IsCommunityInteraction(),
    currentIslandID = GetCurrentIslandID(),
    inputPressure = 0f
};
```

```
        JobHandle optimizationHandle = optimizationJob.Schedule(inputCount, 8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessOptimizationResults(valid, processed, priorities);
        ProcessCulturalValidation(prayerRestrictions, gestureRespect,
sensitivityScores, inputModes);

        // Cleanup
        positions.Dispose();
        pressures.Dispose();
        active.Dispose();
        timestamps.Dispose();
        valid.Dispose();
        processed.Dispose();
        priorities.Dispose();
        prayerRestrictions.Dispose();
        gestureRespect.Dispose();
        traditionalMethods.Dispose();
        communityInput.Dispose();
        religiousSensitivity.Dispose();
        sensitivityScores.Dispose();
        inputAppropriate.Dispose();
        inputModes.Dispose();

        SetupDefaultKeyBindings();
        StartInputMonitoring();
    }
```

```
    void ProcessOptimizationResults(NativeArray<bool> valid, NativeArray<float>
processed, NativeArray<int> priorities)
    {
        int validCount = 0;
        for (int i = 0; i < valid.Length; i++)
        {
            if (valid[i]) validCount++;
        }

        Debug.Log($"Input optimization: {validCount} valid inputs detected");
    }

    void ProcessCulturalValidation(NativeArray<bool> prayer, NativeArray<bool>
gesture, NativeArray<float> sensitivity, NativeArray<int> modes)
    {
        int restrictedCount = 0;
        for (int i = 0; i < prayer.Length; i++)
        {
            if (prayer[i]) restrictedCount++;
        }

        int respectfulCount = 0;
        for (int i = 0; i < gesture.Length; i++)
        {
            if (gesture[i]) respectfulCount++;
        }

        Debug.Log($"Cultural validation: {restrictedCount} prayer restrictions,
{respectfulCount} gesture respects");
    }
```

```
void StartInputMonitoring()
{
    InvokeRepeating("UpdateInput", 0f, 0.02f); // 50 FPS input update
    InvokeRepeating("CheckCulturalContext", 1f, 5f);
    InvokeRepeating("ValidateInputRespect", 2f, 10f);
}

void Update()
{
    if (!gameManager) return;

    UpdateInputDetection();
    CheckCulturalContext();
}

void UpdateInput()
{
    ProcessInput();
    ValidateInputRespect();
}

void UpdateInputDetection()
{
    // Clear previous frame input
    movementInput = Vector2.zero;
    cameraInput = Vector2.zero;
    isRunning = false;
    jumpPressed = false;
    interactPressed = false;
    zoomInput = 0f;
    isUIInputActive = false;
```

```
    // Process different input types
    if (enableKeyboard && !optimizeForMobile)
    {
        ProcessKeyboardInput();
    }

    if (enableMouse && !optimizeForMobile)
    {
        ProcessMouseInput();
    }

    if (enableGamepad && !optimizeForMobile)
    {
        ProcessGamepadInput();
    }

    if (enableTouch && (optimizeForMobile || Input.touchCount > 0))
    {
        ProcessTouchInput();
    }

    if (enableGestures)
    {
        ProcessGestureInput();
    }

    // Apply cultural input restrictions
    ApplyCulturalRestrictions();
}
```

```
    void ProcessKeyboardInput()
    {
        // Movement
        Vector2 keyboardMovement = new Vector2(Input.GetAxis("Horizontal"),
Input.GetAxis("Vertical"));
        movementInput += keyboardMovement;

        // Camera
        if (Input.GetMouseButton(1)) // Right mouse button
        {
            float mouseX = Input.GetAxis("Mouse X");
            float mouseY = Input.GetAxis("Mouse Y");
            cameraInput += new Vector2(mouseX, mouseY);
        }

        // Actions
        isRunning = Input.GetKey(KeyCode.LeftShift) ||
Input.GetKey(KeyCode.RightShift);
        jumpPressed = Input.GetKeyDown(KeyCode.Space);
        interactPressed = Input.GetKeyDown(KeyCode.E);

        // Zoom
        zoomInput = Input.GetAxis("Mouse ScrollWheel");
    }

    void ProcessMouseInput()
    {
        // UI input detection
        if (Input.GetMouseButton(0) || Input.GetMouseButton(1) ||
Input.GetMouseButton(2))
        {
```

```
            isUIInputActive = IsPointerOverUI(Input.mousePosition);
        }
    }

    void ProcessGamepadInput()
    {
        // Gamepad movement
        Vector2 gamepadMovement = new Vector2(Input.GetAxis("Horizontal"),
Input.GetAxis("Vertical"));
        movementInput += gamepadMovement * 0.7f; // Reduce gamepad influence

        // Gamepad camera
        Vector2 gamepadCamera = new
Vector2(Input.GetAxis("RightStickHorizontal"),
Input.GetAxis("RightStickVertical"));
        cameraInput += gamepadCamera * 2f;

        // Gamepad actions
        isRunning = isRunning || Input.GetButton("Run");
        jumpPressed = jumpPressed || Input.GetButtonDown("Jump");
        interactPressed = interactPressed || Input.GetButtonDown("Interact");
    }

    void ProcessTouchInput()
    {
        activeTouches.Clear();

        for (int i = 0; i < Input.touchCount; i++)
        {
            Touch touch = Input.GetTouch(i);
            activeTouches.Add(touch);
```

```
        // Process touch based on phase
        switch (touch.phase)
        {
            case TouchPhase.Began:
                HandleTouchBegan(touch);
                break;

            case TouchPhase.Moved:
                HandleTouchMoved(touch);
                break;

            case TouchPhase.Ended:
                HandleTouchEnded(touch);
                break;
        }
    }

    // Mobile-specific input
    if (optimizeForMobile)
    {
        ProcessMobileInput();
    }
}

void HandleTouchBegan(Touch touch)
{
    // Check if touch is on UI
    if (IsPointerOverUI(touch.position))
    {
        isUIInputActive = true;
```

```
        uiInputPosition = touch.position;
    }
}

void HandleTouchMoved(Touch touch)
{
    // Camera control with touch
    if (!isUIInputActive && touch.phase == TouchPhase.Moved)
    {
        Vector2 deltaPosition = touch.deltaPosition;
        cameraInput += deltaPosition * 0.01f; // Scale down touch sensitivity
    }
}

void HandleTouchEnded(Touch touch)
{
    // Reset UI input
    if (isUIInputActive)
    {
        isUIInputActive = false;
    }
}

void ProcessMobileInput()
{
    // Virtual joystick input (would be connected to UI joystick)
    // Simplified mobile movement
    if (movementInput.magnitude < 0.1f)
    {
        // Use on-screen controls if no other input
        movementInput = GetVirtualJoystickInput();
```

```
        }
    }

    Vector2 GetVirtualJoystickInput()
    {
        // Would get input from virtual joystick UI
        return Vector2.zero;
    }

    void ProcessGestureInput()
    {
        if (disableComplexGestures && optimizeForMobile) return;

        // Detect simple gestures
        if (activeTouches.Count == 2)
        {
            // Pinch zoom
            Touch touch1 = activeTouches[0];
            Touch touch2 = activeTouches[1];

            Vector2 prevPos1 = touch1.position - touch1.deltaPosition;
            Vector2 prevPos2 = touch2.position - touch2.deltaPosition;

            float prevDistance = Vector2.Distance(prevPos1, prevPos2);
            float currentDistance = Vector2.Distance(touch1.position,
touch2.position);

            zoomInput = (currentDistance - prevDistance) * 0.01f;
        }
    }
```

```
void ApplyCulturalRestrictions()
{
    // Apply cultural input restrictions
    InputMode mode = GetInputMode();

    switch (mode)
    {
        case InputMode.Prayer:
            ApplyPrayerRestrictions();
            break;

        case InputMode.Respectful:
            ApplyRespectfulRestrictions();
            break;

        case InputMode.Restricted:
            ApplyRestrictedRestrictions();
            break;
    }
}

void ApplyPrayerRestrictions()
{
    // Severely limit input during prayer time
    movementInput *= prayerTimeInputMultiplier;
    cameraInput *= prayerTimeInputMultiplier;

    // Disable running
    isRunning = false;

    // Limit zoom
```

```
    zoomInput = 0f;
}

void ApplyRespectfulRestrictions()
{
    // Moderate input restrictions
    movementInput *= 0.7f;
    cameraInput *= 0.8f;
}

void ApplyRestrictedRestrictions()
{
    // Block most input
    movementInput = Vector2.zero;
    cameraInput = Vector2.zero;
    isRunning = false;
    jumpPressed = false;
    interactPressed = false;
    zoomInput = 0f;
}

void CheckCulturalContext()
{
    // Update cultural context
    bool newPrayerTime = IsPrayerTime();

    if (newPrayerTime != isPrayerTime)
    {
        isPrayerTime = newPrayerTime;
        OnPrayerTimeChanged(isPrayerTime);
    }
```

```
}

void OnPrayerTimeChanged(bool isPrayer)
{
    if (respectPrayerTimes)
    {
        // Input restrictions will be applied in next frame
        Debug.Log($"Input: Prayer time {(isPrayer ? "started" : "ended")}");
    }
}

void ValidateInputRespect()
{
    // Validate that input is culturally respectful
    if (maintainRespectfulInput)
    {
        bool isRespectfulTime = !IsPrayerTime();

        if (!isRespectfulTime && GetInputMode() != InputMode.Prayer)
        {
            // Force prayer input mode
            // This will be handled in next frame
        }
    }
}

bool IsPointerOverUI(Vector2 position)
{
    // Check if pointer is over UI element
    return UnityEngine.EventSystems.EventSystem.current != null &&
```

```
UnityEngine.EventSystems.EventSystem.current.IsPointerOverGameObject();
    }

    public Vector2 GetMovementInput()
    {
        return movementInput;
    }

    public Vector2 GetCameraInput()
    {
        return cameraInput;
    }

    public bool IsRunning()
    {
        return isRunning;
    }

    public bool IsJumpPressed()
    {
        return jumpPressed;
    }

    public bool IsInteractPressed()
    {
        return interactPressed;
    }

    public float GetZoomInput()
    {
```

```
    return zoomInput;
}

public bool IsUIInputActive()
{
    return isUIInputActive;
}

public Vector2 GetUIInputPosition()
{
    return uiInputPosition;
}

InputMode GetInputMode()
{
    if (respectPrayerTimes && IsPrayerTime())
    {
        return InputMode.Prayer;
    }

    if (IsReligiousContext())
    {
        return InputMode.Respectful;
    }

    if (IsCommunityInteraction())
    {
        return InputMode.Community;
    }

    return InputMode.Normal;
```

```
    }

    void SetupDefaultKeyBindings()
    {
        // Setup default key bindings
        keyBindings["MoveForward"] = KeyCode.W;
        keyBindings["MoveBackward"] = KeyCode.S;
        keyBindings["MoveLeft"] = KeyCode.A;
        keyBindings["MoveRight"] = KeyCode.D;
        keyBindings["Run"] = KeyCode.LeftShift;
        keyBindings["Jump"] = KeyCode.Space;
        keyBindings["Interact"] = KeyCode.E;
        keyBindings["Pause"] = KeyCode.Escape;
    }

    public void SetKeyBinding(string action, KeyCode key)
    {
        keyBindings[action] = key;
    }

    public KeyCode GetKeyBinding(string action)
    {
        return keyBindings.ContainsKey(action) ? keyBindings[action] :
KeyCode.None;
    }

    public void EnableRespectfulMode(bool enabled)
    {
        respectPrayerTimes = enabled;
        maintainRespectfulInput = enabled;
    }
```

```
float GetPrayerTimeWeight()
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0f;
}

bool IsPrayerTime()
{
    return GetPrayerTimeWeight() > 0.7f;
}

bool IsReligiousContext()
{
    return GetPrayerTimeWeight() > 0.5f;
}

bool IsCommunityInteraction()
{
    // Simplified check
    return Time.time % 86400 > 43200 && Time.time % 86400 < 46800;
}

int GetCurrentIslandID()
{
    // Get current island ID from game state
    return 0; // Placeholder
}
```

```
    public override void OnStateChanged(GameState newState, GameState
oldState)
    {
        base.OnStateChanged(newState, oldState);

        switch (newState)
        {
            case GameState.PrayerTime:
                // Input restrictions will be applied automatically
                break;

            case GameState.Paused:
                // Disable all input when paused
                break;

            case GameState.Playing:
                // Enable normal input
                break;
        }
    }

    public InputSnapshot GetInputSnapshot()
    {
        return new InputSnapshot
        {
            movement_input = movementInput,
            camera_input = cameraInput,
            is_running = isRunning,
            jump_pressed = jumpPressed,
            interact_pressed = interactPressed,
```

```
            ui_input_active = isUIInputActive,
            current_mode = GetInputMode().ToString(),
            prayer_time_active = IsPrayerTime(),
            active_touches = activeTouches.Count
        };
    }

    [System.Serializable]
    public class InputSnapshot
    {
        public Vector2 movement_input;
        public Vector2 camera_input;
        public bool is_running;
        public bool jump_pressed;
        public bool interact_pressed;
        public bool ui_input_active;
        public string current_mode;
        public bool prayer_time_active;
        public int active_touches;
    }
}
```

53. AssetManager.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
```

```
using Unity.Mathematics;
using System.Collections.Generic;
using System.IO;

[BurstCompile]
public class AssetManager : GameSystem
{
    [BurstCompile]
    struct AssetLoadingOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> assetSizes;
        [ReadOnly] public NativeArray<int> assetPriorities;
        [ReadOnly] public NativeArray<bool> isEssential;
        [ReadOnly] public NativeArray<float> lastAccessTimes;
        [WriteOnly] public NativeArray<bool> shouldLoadAsset;
        [WriteOnly] public NativeArray<int> loadingOrder;
        [WriteOnly] public NativeArray<float> loadingScores;

        public float availableMemory;
        public float bandwidthLimit;
        public int maxConcurrentLoads;
        public float currentTime;
        public float memoryBudget;

        public void Execute(int index)
        {
            float size = assetSizes[index];
            int priority = assetPriorities[index];
            bool essential = isEssential[index];
            float lastAccess = lastAccessTimes[index];
```

```
            bool shouldLoad = ShouldLoadAsset(size, priority, essential, lastAccess);
            int order = CalculateLoadingOrder(priority, essential, lastAccess);
            float score = CalculateLoadingScore(size, priority, essential, lastAccess);

            shouldLoadAsset[index] = shouldLoad;
            loadingOrder[index] = order;
            loadingScores[index] = score;
        }

        [BurstCompile]
        bool ShouldLoadAsset(float size, int priority, bool essential, float lastAccess)
        {
            bool sizeOk = size < availableMemory * 0.1f;
            bool priorityOk = priority > 5 || essential;
            bool memoryOk = size < memoryBudget * 0.05f;

            return sizeOk && priorityOk && memoryOk;
        }

        [BurstCompile]
        int CalculateLoadingOrder(int priority, bool essential, float lastAccess)
        {
            int order = priority * 100;
            if (essential) order -= 1000;

            float recencyBonus = (currentTime - lastAccess) < 60f ? -50 : 0; // Recent
access bonus
            order += (int)recencyBonus;

            return math.max(0, order);
        }
```

```
        [BurstCompile]
        float CalculateLoadingScore(float size, int priority, bool essential, float
lastAccess)
        {
            float sizeScore = math.saturate(1.0f - (size / 100f));
            float priorityScore = priority / 10f;
            float essentialBonus = essential ? 0.3f : 0f;
            float recencyScore = math.saturate(1.0f - ((currentTime - lastAccess) /
3600f));

            return (sizeScore + priorityScore + essentialBonus + recencyScore) *
0.25f;
        }
    }

    [BurstCompile]
    struct MaldivianCulturalAssets : IJob
    {
        public NativeArray<bool> culturalAssetAuthenticity;
        public NativeArray<bool> religiousAssetRespect;
        public NativeArray<bool> traditionalPracticeAssets;
        public NativeArray<bool> communityAssetAppropriate;
        public NativeArray<bool> environmentalAssetSensitivity;

        public NativeArray<float> culturalAuthenticityScores;
        public NativeArray<bool> isAssetLoadingRespectful;
        public NativeArray<int> appropriateAssetLoadingModes;

        public int currentIslandID;
        public bool isPrayerTime;
```

```
public bool isCulturalEvent;
public bool isReligiousHoliday;
public float prayerTimeWeight;

public void Execute()
{
    ValidateCulturalAssetAuthenticity();
    ValidateReligiousAssetRespect();
    ValidateTraditionalPracticeAssets();
    CalculateAssetLoadingRespectfulness();
    DetermineAppropriateAssetLoadingMode();
}

[BurstCompile]
void ValidateCulturalAssetAuthenticity()
{
    for (int i = 0; i < culturalAssetAuthenticity.Length; i++)
    {
        bool isAuthentic = currentIslandID >= 0 && currentIslandID < 41;
        culturalAssetAuthenticity[i] = isAuthentic;

        if (isAuthentic)
        {
            culturalAuthenticityScores[i] = 0.95f;
        }
    }
}

[BurstCompile]
void ValidateReligiousAssetRespect()
{
```

```
    for (int i = 0; i < religiousAssetRespect.Length; i++)
    {
        bool isRespectful = !isPrayerTime && prayerTimeWeight < 0.5f;
        religiousAssetRespect[i] = isRespectful;

        if (isRespectful)
        {
            isAssetLoadingRespectful[i] = true;
        }
    }
}

[BurstCompile]
void ValidateTraditionalPracticeAssets()
{
    for (int i = 0; i < traditionalPracticeAssets.Length; i++)
    {
        bool isTraditional = currentIslandID >= 0 && currentIslandID < 41;
        traditionalPracticeAssets[i] = isTraditional;

        if (isTraditional)
        {
            culturalAuthenticityScores[religiousAssetRespect.Length + i] = 0.90f;
        }
    }
}

[BurstCompile]
void CalculateAssetLoadingRespectfulness()
{
    float totalScore = 0.0f;
```

```
    int totalChecks = 0;

    for (int i = 0; i < culturalAssetAuthenticity.Length; i++)
    {
        if (culturalAssetAuthenticity[i]) totalScore += 1.0f;
        totalChecks++;
    }

    for (int i = 0; i < religiousAssetRespect.Length; i++)
    {
        if (religiousAssetRespect[i]) totalScore += 1.0f;
        totalChecks++;
    }

    float averageScore = totalChecks > 0 ? totalScore / totalChecks : 0.0f;

    for (int i = 0; i < isAssetLoadingRespectful.Length; i++)
    {
        isAssetLoadingRespectful[i] = averageScore > 0.8f;
    }
}

[BurstCompile]
void DetermineAppropriateAssetLoadingMode()
{
    int mode = 1; // Normal loading

    if (isPrayerTime) mode = 2; // Restricted loading
    if (isReligiousHoliday) mode = 3; // Holiday-aware loading
    if (isCulturalEvent) mode = 4; // Event-aware loading
```

```
        for (int i = 0; i < appropriateAssetLoadingModes.Length; i++)
        {
            appropriateAssetLoadingModes[i] = mode;
        }
    }
}

public static AssetManager Instance { get; private set; }

[Header("Asset Configuration")]
public string assetsPath = "Assets/Resources";
public bool enableAsyncLoading = true;
public bool enableCaching = true;
public int maxCacheSize = 1000; // MB
public int maxConcurrentLoads = 3;

[Header("Cultural Asset Settings")]
public bool prioritizeCulturalAssets = true;
public bool respectPrayerTimeLoading = true;
public bool maintainAuthenticity = true;
public bool enableRespectfulLoading = true;

[Header("Mobile Optimization")]
public bool optimizeForMobile = true;
public int maxMobileAssets = 500;
public bool reduceAssetQuality = false;
public bool enableAssetCompression = true;

private Dictionary<string, Object> loadedAssets;
private Dictionary<string, AssetLoadRequest> loadRequests;
private Queue<AssetLoadRequest> loadQueue;
```

```
private List<string> loadingAssets;
private float currentCacheUsage;
private bool isLoading = false;

public enum AssetType
{
    Texture = 0,
    Model = 1,
    Audio = 2,
    Prefab = 3,
    Material = 4,
    ScriptableObject = 5,
    Cultural = 6,
    Religious = 7,
    Traditional = 8
}

public class AssetLoadRequest
{
    public string assetName;
    public AssetType assetType;
    public int priority;
    public bool isEssential;
    public bool isCultural;
    public Action<Object> onComplete;
    public Action<string> onError;
    public float estimatedSize;
    public string culturalContext;
}

public override void Initialize(MainGameManager manager)
```

```
    {
        base.Initialize(manager);
        InitializeAssetManager();
    }

    void InitializeAssetManager()
    {
        // Initialize collections
        loadedAssets = new Dictionary<string, Object>();
        loadRequests = new Dictionary<string, AssetLoadRequest>();
        loadQueue = new Queue<AssetLoadRequest>();
        loadingAssets = new List<string>();

        // Initialize asset loading optimization
        int assetCount = 100;
        var sizes = new NativeArray<float>(assetCount, Allocator.TempJob);
        var priorities = new NativeArray<int>(assetCount, Allocator.TempJob);
        var essential = new NativeArray<bool>(assetCount, Allocator.TempJob);
        var accessTimes = new NativeArray<float>(assetCount,
Allocator.TempJob);
        var shouldLoad = new NativeArray<bool>(assetCount, Allocator.TempJob);
        var order = new NativeArray<int>(assetCount, Allocator.TempJob);
        var scores = new NativeArray<float>(assetCount, Allocator.TempJob);

        // Initialize with sample asset data
        for (int i = 0; i < assetCount; i++)
        {
            sizes[i] = UnityEngine.Random.Range(1f, 50f); // 1-50 MB
            priorities[i] = UnityEngine.Random.Range(1, 10);
            essential[i] = i < 30; // First 30 are essential
```

```
            accessTimes[i] = Time.time - UnityEngine.Random.Range(0f, 3600f); // Last hour
        }

        var optimizationJob = new AssetLoadingOptimization
        {
            assetSizes = sizes,
            assetPriorities = priorities,
            isEssential = essential,
            lastAccessTimes = accessTimes,
            shouldLoadAsset = shouldLoad,
            loadingOrder = order,
            loadingScores = scores,
            availableMemory = SystemInfo.systemMemorySize,
            bandwidthLimit = 10f * 1024f * 1024f, // 10 MB/s
            maxConcurrentLoads = maxConcurrentLoads,
            currentTime = Time.time,
            memoryBudget = maxCacheSize * 1024f * 1024f
        };

        // Initialize cultural asset validation
        int culturalAssets = 50;
        int religiousAssets = 25;
        int traditionalAssets = 30;
        int communityAssets = 20;
        int environmentalAssets = 40;

        var culturalAssetsArray = new NativeArray<bool>(culturalAssets, Allocator.TempJob);
        var religiousAssetsArray = new NativeArray<bool>(religiousAssets, Allocator.TempJob);
```

```
        var traditionalAssetsArray = new NativeArray<bool>(traditionalAssets,
Allocator.TempJob);
        var communityAssetsArray = new NativeArray<bool>(communityAssets,
Allocator.TempJob);
        var environmentalAssetsArray = new
NativeArray<bool>(environmentalAssets, Allocator.TempJob);
        var authenticityScores = new NativeArray<float>(culturalAssets +
religiousAssets + traditionalAssets + communityAssets + environmentalAssets,
Allocator.TempJob);
        var respectfulLoading = new NativeArray<bool>(culturalAssets +
religiousAssets + traditionalAssets + communityAssets + environmentalAssets,
Allocator.TempJob);
        var loadingModes = new NativeArray<int>(culturalAssets + religiousAssets
+ traditionalAssets + communityAssets + environmentalAssets,
Allocator.TempJob);

        // Initialize with sample cultural data
        for (int i = 0; i < culturalAssets; i++) culturalAssetsArray[i] = true;
        for (int i = 0; i < religiousAssets; i++) religiousAssetsArray[i] = true;
        for (int i = 0; i < traditionalAssets; i++) traditionalAssetsArray[i] = true;
        for (int i = 0; i < communityAssets; i++) communityAssetsArray[i] = true;
        for (int i = 0; i < environmentalAssets; i++) environmentalAssetsArray[i] =
true;

        var culturalJob = new MaldivianCulturalAssets
        {
            culturalAssetAuthenticity = culturalAssetsArray,
            religiousAssetRespect = religiousAssetsArray,
            traditionalPracticeAssets = traditionalAssetsArray,
            communityAssetAppropriate = communityAssetsArray,
            environmentalAssetSensitivity = environmentalAssetsArray,
```

```
            culturalAuthenticityScores = authenticityScores,
            isAssetLoadingRespectful = respectfulLoading,
            appropriateAssetLoadingModes = loadingModes,
            currentIslandID = 0,
            isPrayerTime = IsPrayerTime(),
            isCulturalEvent = IsCulturalEvent(),
            isReligiousHoliday = IsReligiousHoliday(),
            prayerTimeWeight = GetPrayerTimeWeight()
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(assetCount, 8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessOptimizationResults(shouldLoad, order, scores);
        ProcessCulturalValidation(culturalAssetsArray, religiousAssetsArray,
authenticityScores, loadingModes);

        // Cleanup
        sizes.Dispose();
        priorities.Dispose();
        essential.Dispose();
        accessTimes.Dispose();
        shouldLoad.Dispose();
        order.Dispose();
        scores.Dispose();
        culturalAssetsArray.Dispose();
        religiousAssetsArray.Dispose();
        traditionalAssetsArray.Dispose();
        communityAssetsArray.Dispose();
```

```
        environmentalAssetsArray.Dispose();
        authenticityScores.Dispose();
        respectfulLoading.Dispose();
        loadingModes.Dispose();

        StartAssetMonitoring();
    }

    void ProcessOptimizationResults(NativeArray<bool> shouldLoad,
NativeArray<int> order, NativeArray<float> scores)
    {
        int loadCount = 0;
        float avgScore = 0f;

        for (int i = 0; i < shouldLoad.Length; i++)
        {
            if (shouldLoad[i]) loadCount++;
            avgScore += scores[i];
        }

        avgScore /= shouldLoad.Length;

        Debug.Log($"Asset optimization: {loadCount} assets should load, average
score: {avgScore:F2}");
    }

    void ProcessCulturalValidation(NativeArray<bool> cultural, NativeArray<bool>
religious, NativeArray<float> authenticity, NativeArray<int> modes)
    {
        int authenticCount = 0;
        for (int i = 0; i < cultural.Length; i++)
```

```
        {
            if (cultural[i]) authenticCount++;
        }

        int respectfulCount = 0;
        for (int i = 0; i < religious.Length; i++)
        {
            if (religious[i]) respectfulCount++;
        }

        Debug.Log($"Cultural validation: {authenticCount} authentic cultural assets,
{respectfulCount} respectful religious assets");
    }

    void StartAssetMonitoring()
    {
        InvokeRepeating("ProcessAssetQueue", 0.5f, 0.1f);
        InvokeRepeating("MonitorActiveLoads", 0f, 0.05f);
        InvokeRepeating("ValidateCulturalLoading", 5f, 15f);
    }

    public void LoadAsset(string assetName, AssetType assetType,
Action<Object> onComplete, int priority = 5)
    {
        // Check if asset is already loaded
        if (loadedAssets.ContainsKey(assetName))
        {
            onComplete?.Invoke(loadedAssets[assetName]);
            return;
        }
```

```
    // Check if asset is currently loading
    if (loadRequests.ContainsKey(assetName))
    {
        // Add callback to existing request
        var existingRequest = loadRequests[assetName];
        existingRequest.onComplete += onComplete;
        return;
    }

    // Create new load request
    AssetLoadRequest request = new AssetLoadRequest
    {
        assetName = assetName,
        assetType = assetType,
        priority = priority,
        isEssential = priority > 7,
        isCultural = IsCulturalAsset(assetName),
        onComplete = onComplete,
        estimatedSize = EstimateAssetSize(assetType),
        culturalContext = GetCulturalContext(assetName)
    };

    // Check if we can load respectfully
    if (respectPrayerTimeLoading && !CanLoadRespectfully(request))
    {
        QueueLoadForLater(request);
        return;
    }

    QueueAssetRequest(request);
}
```

```
    bool CanLoadRespectfully(AssetLoadRequest request)
    {
        if (!enableRespectfulLoading) return true;

        // Check if it's prayer time
        if (IsPrayerTime() && !request.isEssential)
        {
            return false;
        }

        // Check if it's a religious holiday
        if (IsReligiousHoliday() && request.isCultural)
        {
            return false;
        }

        return true;
    }

    void QueueLoadForLater(AssetLoadRequest request)
    {
        // Queue the load for a more respectful time
        StartCoroutine(WaitForRespectfulTime(request));
    }

    System.Collections.IEnumerator WaitForRespectfulTime(AssetLoadRequest
request)
    {
        while (!CanLoadRespectfully(request))
        {
```

```
            yield return new WaitForSeconds(1f);
        }

        QueueAssetRequest(request);
    }

    void QueueAssetRequest(AssetLoadRequest request)
    {
        loadRequests[request.assetName] = request;

        // Insert into queue based on priority
        List<AssetLoadRequest> tempList = new
List<AssetLoadRequest>(loadQueue.ToArray());
        tempList.Add(request);
        tempList.Sort((a, b) => b.priority.CompareTo(a.priority));

        loadQueue.Clear();
        foreach (var item in tempList)
        {
            loadQueue.Enqueue(item);
        }
    }

    void ProcessAssetQueue()
    {
        if (isLoading || loadQueue.Count == 0) return;
        if (loadingAssets.Count >= maxConcurrentLoads) return;

        AssetLoadRequest request = loadQueue.Dequeue();
        StartCoroutine(LoadAssetAsync(request));
    }
```

```
System.Collections.IEnumerator LoadAssetAsync(AssetLoadRequest request)
{
    isLoading = true;
    loadingAssets.Add(request.assetName);

    // Simulate loading delay (in production, would use Resources.LoadAsync)
    yield return new WaitForSeconds(0.1f);

    // Load asset based on type
    Object loadedAsset = null;

    switch (request.assetType)
    {
        case AssetType.Texture:
            loadedAsset = Resources.Load<Texture2D>(request.assetName);
            break;

        case AssetType.Model:
            loadedAsset = Resources.Load<GameObject>(request.assetName);
            break;

        case AssetType.Audio:
            loadedAsset = Resources.Load<AudioClip>(request.assetName);
            break;

        case AssetType.Prefab:
            loadedAsset = Resources.Load<GameObject>(request.assetName);
            break;

        case AssetType.Material:
```

```
            loadedAsset = Resources.Load<Material>(request.assetName);
            break;

        default:
            loadedAsset = Resources.Load(request.assetName);
            break;
    }

    if (loadedAsset != null)
    {
        loadedAssets[request.assetName] = loadedAsset;
        currentCacheUsage += request.estimatedSize;

        request.onComplete?.Invoke(loadedAsset);
    }
    else
    {
        request.onError?.Invoke($"Failed to load asset: {request.assetName}");
    }

    loadingAssets.Remove(request.assetName);
    loadRequests.Remove(request.assetName);
    isLoading = false;
}

void MonitorActiveLoads()
{
    // Clean up completed loads
    // Implementation would track actual async operations
}
```

```
public T GetAsset<T>(string assetName) where T : Object
{
    if (loadedAssets.ContainsKey(assetName))
    {
        return loadedAssets[assetName] as T;
    }

    return null;
}

public void UnloadAsset(string assetName)
{
    if (loadedAssets.ContainsKey(assetName))
    {
        // Estimate size reduction
        if (loadedAssets[assetName] is Texture2D)
        {
            currentCacheUsage -= 5f; // Estimated texture size
        }
        else if (loadedAssets[assetName] is AudioClip)
        {
            currentCacheUsage -= 2f; // Estimated audio size
        }

        Resources.UnloadAsset(loadedAssets[assetName]);
        loadedAssets.Remove(assetName);
    }
}

public void UnloadUnusedAssets()
{
```

```
    if (currentCacheUsage > maxCacheSize * 0.9f)
    {
        // Unload least recently used assets
        UnloadLeastUsedAssets();
    }
}

void UnloadLeastUsedAssets()
{
    // Simple LRU implementation
    List<string> assetsToUnload = new List<string>();

    // Find assets to unload (simplified)
    foreach (var kvp in loadedAssets)
    {
        if (Random.Range(0, 100) < 20) // 20% chance to unload
        {
            assetsToUnload.Add(kvp.Key);
        }
    }

    foreach (string assetName in assetsToUnload)
    {
        UnloadAsset(assetName);
    }
}

bool IsCulturalAsset(string assetName)
{
    // Check if asset contains cultural content
    return assetName.Contains("Cultural") ||
```

```
            assetName.Contains("Traditional") ||
            assetName.Contains("Religious") ||
            assetName.Contains("Mosque") ||
            assetName.Contains("Prayer");
}

float EstimateAssetSize(AssetType type)
{
    return type switch
    {
        AssetType.Texture => 5f, // 5 MB
        AssetType.Model => 10f, // 10 MB
        AssetType.Audio => 2f, // 2 MB
        AssetType.Prefab => 1f, // 1 MB
        AssetType.Material => 0.5f, // 0.5 MB
        AssetType.Cultural => 8f, // Cultural assets are larger
        AssetType.Religious => 3f, // Religious assets
        AssetType.Traditional => 6f, // Traditional assets
        _ => 2f
    };
}

string GetCulturalContext(string assetName)
{
    if (assetName.Contains("Prayer")) return "Religious";
    if (assetName.Contains("Mosque")) return "Religious";
    if (assetName.Contains("Cultural")) return "Cultural";
    if (assetName.Contains("Traditional")) return "Traditional";
    return "General";
}
```

```
void ValidateCulturalLoading()
{
    // Validate that asset loading respects cultural requirements
    if (maintainAuthenticity)
    {
        bool isRespectfulTime = !IsPrayerTime();

        if (!isRespectfulTime && loadQueue.Count > 0)
        {
            // Pause non-essential loading
            Debug.Log("Asset loading paused for cultural respect");
        }
    }
}

float GetPrayerTimeWeight()
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0f;
}

bool IsPrayerTime()
{
    return GetPrayerTimeWeight() > 0.7f;
}

bool IsCulturalEvent()
{
```

```
        // Simplified check
        return Time.time % 86400 > 43200 && Time.time % 86400 < 46800;
    }

    bool IsReligiousHoliday()
    {
        // Simplified check - would check religious calendar
        return DateTime.Now.DayOfWeek == DayOfWeek.Friday;
    }

    public override void OnStateChanged(GameState newState, GameState
oldState)
    {
        base.OnStateChanged(newState, oldState);

        switch (newState)
        {
            case GameState.Loading:
                // Prioritize loading screen assets
                break;

            case GameState.PrayerTime:
                // Pause non-essential asset loading
                if (respectPrayerTimeLoading)
                {
                    // Clear non-essential loads from queue
                }
                break;
        }
    }
```

```
    public AssetSnapshot GetAssetSnapshot()
    {
        return new AssetSnapshot
        {
            loaded_assets = loadedAssets.Count,
            loading_assets = loadingAssets.Count,
            queue_size = loadQueue.Count,
            cache_usage_mb = currentCacheUsage,
            cache_limit_mb = maxCacheSize,
            cultural_priority = prioritizeCulturalAssets,
            prayer_time_respect = respectPrayerTimeLoading
        };
    }

    [System.Serializable]
    public class AssetSnapshot
    {
        public int loaded_assets;
        public int loading_assets;
        public int queue_size;
        public float cache_usage_mb;
        public float cache_limit_mb;
        public bool cultural_priority;
        public bool prayer_time_respect;
    }
}
```

54. QualitySettings.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;

[BurstCompile]
public class QualitySettings : GameSystem
{
    [BurstCompile]
    struct QualityOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> performanceMetrics;
        [ReadOnly] public NativeArray<bool> qualityFeatures;
        [ReadOnly] public NativeArray<int> featurePriorities;
        [WriteOnly] public NativeArray<bool> shouldEnableFeature;
        [WriteOnly] public NativeArray<float> qualityScores;
        [WriteOnly] public NativeArray<int> optimizationRecommendations;

        public float targetFrameTime;
        public float minAcceptableFPS;
        public int targetQualityLevel;
        public float performanceThreshold;

        public void Execute(int index)
        {
            float metric = performanceMetrics[index];
            bool feature = qualityFeatures[index];
            int priority = featurePriorities[index];
```

```
            bool shouldEnable = ShouldEnableQualityFeature(metric, feature,
priority);
            float score = CalculateQualityScore(metric, priority);
            int recommendation = GenerateQualityRecommendation(metric, feature,
priority);

            shouldEnableFeature[index] = shouldEnable;
            qualityScores[index] = score;
            optimizationRecommendations[index] = recommendation;
        }

        [BurstCompile]
        bool ShouldEnableQualityFeature(float metric, bool currentState, int priority)
        {
            bool performanceOk = metric < performanceThreshold;
            bool priorityOk = priority > 5 || currentState;

            return performanceOk && priorityOk;
        }

        [BurstCompile]
        float CalculateQualityScore(float metric, int priority)
        {
            float performanceScore = math.saturate(1.0f - (metric /
targetFrameTime));
            float priorityScore = priority / 10f;

            return (performanceScore + priorityScore) * 0.5f;
        }

        [BurstCompile]
```

```
    int GenerateQualityRecommendation(float metric, bool currentState, int
priority)
    {
        if (metric > targetFrameTime * 1.5f) return 3; // Critical reduction
        if (metric > targetFrameTime * 1.2f) return 2; // Major reduction
        if (metric > targetFrameTime) return 1; // Minor reduction
        return 0; // No change needed
    }
}

[BurstCompile]
struct MaldivianCulturalQuality : IJob
{
    public NativeArray<bool> prayerTimeQualityReduction;
    public NativeArray<bool> culturalSceneFidelity;
    public NativeArray<bool> traditionalDetailPreservation;
    public NativeArray<bool> communityEventOptimization;
    public NativeArray<bool> religiousSiteQuality;

    public NativeArray<float> culturalSensitivityScores;
    public NativeArray<bool> isQualitySettingRespectful;
    public NativeArray<int> appropriateQualityLevels;

    public float currentPrayerTimeWeight;
    public bool isReligiousContext;
    public bool isCommunityEvent;
    public int currentIslandID;
    public float currentPerformance;

    public void Execute()
    {
```

```
    ValidatePrayerTimeQualityReduction();
    ValidateCulturalSceneFidelity();
    ValidateTraditionalDetailPreservation();
    CalculateQualitySettingRespectfulness();
    DetermineAppropriateQualityLevel();
}

[BurstCompile]
void ValidatePrayerTimeQualityReduction()
{
    for (int i = 0; i < prayerTimeQualityReduction.Length; i++)
    {
        bool shouldReduce = currentPrayerTimeWeight > 0.7f;
        prayerTimeQualityReduction[i] = shouldReduce;

        if (shouldReduce)
        {
            culturalSensitivityScores[i] = 0.95f;
        }
    }
}

[BurstCompile]
void ValidateCulturalSceneFidelity()
{
    for (int i = 0; i < culturalSceneFidelity.Length; i++)
    {
        bool shouldPreserve = isReligiousContext || isCommunityEvent;
        culturalSceneFidelity[i] = shouldPreserve;

        if (shouldPreserve)
```

```
            {
                isQualitySettingRespectful[i] = true;
            }
        }
    }

    [BurstCompile]
    void ValidateTraditionalDetailPreservation()
    {
        for (int i = 0; i < traditionalDetailPreservation.Length; i++)
        {
            bool isTraditional = currentIslandID >= 0 && currentIslandID < 41;
            traditionalDetailPreservation[i] = isTraditional;

            if (isTraditional)
            {
                culturalSensitivityScores[prayerTimeQualityReduction.Length + i] =
0.90f;
            }
        }
    }

    [BurstCompile]
    void CalculateQualitySettingRespectfulness()
    {
        float totalScore = 0.0f;
        int totalChecks = 0;

        for (int i = 0; i < prayerTimeQualityReduction.Length; i++)
        {
            if (prayerTimeQualityReduction[i]) totalScore += 1.0f;
```

```
        totalChecks++;
    }

    for (int i = 0; i < culturalSceneFidelity.Length; i++)
    {
        if (culturalSceneFidelity[i]) totalScore += 1.0f;
        totalChecks++;
    }

    float averageScore = totalChecks > 0 ? totalScore / totalChecks : 0.0f;

    for (int i = 0; i < isQualitySettingRespectful.Length; i++)
    {
        isQualitySettingRespectful[i] = averageScore > 0.8f;
    }
}

[BurstCompile]
void DetermineAppropriateQualityLevel()
{
    int level = 2; // Medium quality

    if (currentPrayerTimeWeight > 0.7f) level = 0; // Low quality
    if (isReligiousContext) level = 1; // Low-medium quality
    if (isCommunityEvent) level = 3; // High quality
    if (currentPerformance > 60f) level = 4; // Ultra quality

    for (int i = 0; i < appropriateQualityLevels.Length; i++)
    {
        appropriateQualityLevels[i] = level;
    }
```

```
    }
}

public static QualitySettings Instance { get; private set; }

[Header("Quality Configuration")]
public int currentQualityLevel = 2;
public bool autoAdjustQuality = true;
public float qualityCheckInterval = 5f;
public float targetFPS = 60f;
public float minimumAcceptableFPS = 30f;

[Header("Cultural Quality Settings")]
public bool respectPrayerTimes = true;
public bool preserveCulturalFidelity = true;
public bool maintainTraditionalDetails = true;
public float prayerTimeQualityMultiplier = 0.6f;

[Header("Mobile Optimization")]
public bool optimizeForMobile = true;
public int maxMobileQuality = 2;
public bool reduceTextureResolution = false;
public bool disableAdvancedShading = true;

private float currentFPS;
private float averageFrameTime;
private int frameCount;
private float lastQualityCheck;
private bool isPrayerTime;
private QualityProfile[] qualityProfiles;
```

```
public enum QualityProfile
{
    Low = 0,
    Medium = 1,
    High = 2,
    Ultra = 3,
    Custom = 4
}

[System.Serializable]
public class QualityLevel
{
    public string name;
    public int textureResolution = 512;
    public int shadowResolution = 512;
    public bool enableShadows = true;
    public bool enableAntiAliasing = true;
    public int antiAliasingLevel = 2;
    public bool enableBloom = true;
    public bool enableDepthOfField = false;
    public int maxLODLevel = 0;
    public float lodBias = 1.0f;
    public float renderScale = 1.0f;
    public bool enableVSync = true;
    public int maxParticleCount = 1000;
    public bool enableRealTimeReflections = false;
    public int textureQuality = 2;
    public bool preserveCulturalDetails = true;
}

void Awake()
```

```
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }

    public override void Initialize(MainGameManager manager)
    {
        base.Initialize(manager);
        InitializeQualitySettings();
    }

    void InitializeQualitySettings()
    {
        // Initialize quality profiles
        InitializeQualityProfiles();

        // Initialize quality optimization
        int featureCount = 20;
        var metrics = new NativeArray<float>(featureCount, Allocator.TempJob);
        var features = new NativeArray<bool>(featureCount, Allocator.TempJob);
        var priorities = new NativeArray<int>(featureCount, Allocator.TempJob);
        var shouldEnable = new NativeArray<bool>(featureCount,
Allocator.TempJob);
        var scores = new NativeArray<float>(featureCount, Allocator.TempJob);
```

```
        var recommendations = new NativeArray<int>(featureCount,
Allocator.TempJob);

        // Initialize with sample quality data
        for (int i = 0; i < featureCount; i++)
        {
            metrics[i] = UnityEngine.Random.Range(0.008f, 0.02f); // 50-125 FPS
range
            features[i] = UnityEngine.Random.Range(0, 2) == 1;
            priorities[i] = UnityEngine.Random.Range(1, 10);
        }

        var optimizationJob = new QualityOptimization
        {
            performanceMetrics = metrics,
            qualityFeatures = features,
            featurePriorities = priorities,
            shouldEnableFeature = shouldEnable,
            qualityScores = scores,
            optimizationRecommendations = recommendations,
            targetFrameTime = 1.0f / targetFPS,
            minAcceptableFPS = minimumAcceptableFPS,
            targetQualityLevel = currentQualityLevel,
            performanceThreshold = 0.016f
        };

        // Initialize cultural quality validation
        var prayerReduction = new NativeArray<bool>(6, Allocator.TempJob);
        var culturalFidelity = new NativeArray<bool>(8, Allocator.TempJob);
        var traditionalPreservation = new NativeArray<bool>(10,
Allocator.TempJob);
```

```
        var communityOptimization = new NativeArray<bool>(5,
Allocator.TempJob);
        var religiousQuality = new NativeArray<bool>(3, Allocator.TempJob);
        var sensitivityScores = new NativeArray<float>(50, Allocator.TempJob);
        var qualityRespectful = new NativeArray<bool>(50, Allocator.TempJob);
        var qualityLevels = new NativeArray<int>(50, Allocator.TempJob);

        var culturalJob = new MaldivianCulturalQuality
        {
            prayerTimeQualityReduction = prayerReduction,
            culturalSceneFidelity = culturalFidelity,
            traditionalDetailPreservation = traditionalPreservation,
            communityEventOptimization = communityOptimization,
            religiousSiteQuality = religiousQuality,
            culturalSensitivityScores = sensitivityScores,
            isQualitySettingRespectful = qualityRespectful,
            appropriateQualityLevels = qualityLevels,
            currentPrayerTimeWeight = GetPrayerTimeWeight(),
            isReligiousContext = IsReligiousContext(),
            isCommunityEvent = IsCulturalEvent(),
            currentIslandID = GetCurrentIslandID(),
            currentPerformance = targetFPS
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(featureCount,
8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessOptimizationResults(shouldEnable, scores, recommendations);
```

```
        ProcessCulturalValidation(prayerReduction, culturalFidelity,
sensitivityScores, qualityLevels);

        // Cleanup
        metrics.Dispose();
        features.Dispose();
        priorities.Dispose();
        shouldEnable.Dispose();
        scores.Dispose();
        recommendations.Dispose();
        prayerReduction.Dispose();
        culturalFidelity.Dispose();
        traditionalPreservation.Dispose();
        communityOptimization.Dispose();
        religiousQuality.Dispose();
        sensitivityScores.Dispose();
        qualityRespectful.Dispose();
        qualityLevels.Dispose();

        ApplyQualitySettings();
        StartQualityMonitoring();
    }

    void InitializeQualityProfiles()
    {
        qualityProfiles = new QualityLevel[5];

        // Low Quality
        qualityProfiles[0] = new QualityLevel
        {
            name = "Low",
```

```
    textureResolution = 256,
    shadowResolution = 256,
    enableShadows = false,
    enableAntiAliasing = false,
    enableBloom = false,
    enableDepthOfField = false,
    maxLODLevel = 2,
    lodBias = 0.5f,
    renderScale = 0.8f,
    enableVSync = false,
    maxParticleCount = 100,
    enableRealTimeReflections = false,
    textureQuality = 0,
    preserveCulturalDetails = false
};

// Medium Quality
qualityProfiles[1] = new QualityLevel
{
    name = "Medium",
    textureResolution = 512,
    shadowResolution = 512,
    enableShadows = true,
    enableAntiAliasing = true,
    antiAliasingLevel = 2,
    enableBloom = true,
    enableDepthOfField = false,
    maxLODLevel = 1,
    lodBias = 0.8f,
    renderScale = 0.9f,
    enableVSync = true,
```

```
    maxParticleCount = 500,
    enableRealTimeReflections = false,
    textureQuality = 1,
    preserveCulturalDetails = true
};

// High Quality
qualityProfiles[2] = new QualityLevel
{
    name = "High",
    textureResolution = 1024,
    shadowResolution = 1024,
    enableShadows = true,
    enableAntiAliasing = true,
    antiAliasingLevel = 4,
    enableBloom = true,
    enableDepthOfField = true,
    maxLODLevel = 0,
    lodBias = 1.0f,
    renderScale = 1.0f,
    enableVSync = true,
    maxParticleCount = 1000,
    enableRealTimeReflections = true,
    textureQuality = 2,
    preserveCulturalDetails = true
};

// Ultra Quality
qualityProfiles[3] = new QualityLevel
{
    name = "Ultra",
```

```
    textureResolution = 2048,
    shadowResolution = 2048,
    enableShadows = true,
    enableAntiAliasing = true,
    antiAliasingLevel = 8,
    enableBloom = true,
    enableDepthOfField = true,
    maxLODLevel = 0,
    lodBias = 1.5f,
    renderScale = 1.2f,
    enableVSync = true,
    maxParticleCount = 2000,
    enableRealTimeReflections = true,
    textureQuality = 3,
    preserveCulturalDetails = true
};

// Custom Quality (will be modified at runtime)
qualityProfiles[4] = new QualityLevel
{
    name = "Custom",
    textureResolution = 1024,
    shadowResolution = 1024,
    enableShadows = true,
    enableAntiAliasing = true,
    antiAliasingLevel = 4,
    enableBloom = true,
    enableDepthOfField = false,
    maxLODLevel = 0,
    lodBias = 1.0f,
    renderScale = 1.0f,
```

```
            enableVSync = true,
            maxParticleCount = 1000,
            enableRealTimeReflections = false,
            textureQuality = 2,
            preserveCulturalDetails = true
        };
    }

    void ProcessOptimizationResults(NativeArray<bool> shouldEnable,
NativeArray<float> scores, NativeArray<int> recommendations)
    {
        int enableCount = 0;
        float avgScore = 0f;

        for (int i = 0; i < shouldEnable.Length; i++)
        {
            if (shouldEnable[i]) enableCount++;
            avgScore += scores[i];
        }

        avgScore /= shouldEnable.Length;

        Debug.Log($"Quality optimization: {enableCount} features should be
enabled, average score: {avgScore:F2}");
    }

    void ProcessCulturalValidation(NativeArray<bool> prayer, NativeArray<bool>
cultural, NativeArray<float> sensitivity, NativeArray<int> levels)
    {
        int reductionCount = 0;
        for (int i = 0; i < prayer.Length; i++)
```

```
        {
            if (prayer[i]) reductionCount++;
        }

        int fidelityCount = 0;
        for (int i = 0; i < cultural.Length; i++)
        {
            if (cultural[i]) fidelityCount++;
        }

        Debug.Log($"Cultural validation: {reductionCount} prayer reductions,
{fidelityCount} cultural fidelities");
    }

    void StartQualityMonitoring()
    {
        InvokeRepeating("UpdateQualityMetrics", 0f, 0.5f);
        InvokeRepeating("AutoAdjustQuality", 1f, qualityCheckInterval);
        InvokeRepeating("CheckCulturalContext", 2f, 10f);
    }

    void Update()
    {
        if (!gameManager) return;

        UpdateQualityMetrics();
        CheckCulturalContext();
    }

    void UpdateQualityMetrics()
    {
```

```
        // Calculate current FPS
        currentFPS = 1.0f / Time.unscaledDeltaTime;
        frameCount++;

        // Update average frame time
        averageFrameTime += (Time.unscaledDeltaTime - averageFrameTime) /
frameCount;
    }

    void AutoAdjustQuality()
    {
        if (!autoAdjustQuality) return;

        // Check if we need to adjust quality
        if (Time.time - lastQualityCheck > qualityCheckInterval)
        {
            if (currentFPS < minimumAcceptableFPS)
            {
                // Reduce quality
                ReduceQuality();
            }
            else if (currentFPS > targetFPS * 1.2f && currentQualityLevel <
qualityProfiles.Length - 1)
            {
                // Increase quality if performance allows
                IncreaseQuality();
            }

            lastQualityCheck = Time.time;
        }
    }
```

```
void ReduceQuality()
{
    if (currentQualityLevel > 0)
    {
        currentQualityLevel--;
        ApplyQualitySettings();
        Debug.Log($"Quality reduced to level: {currentQualityLevel}");
    }
}

void IncreaseQuality()
{
    if (currentQualityLevel < qualityProfiles.Length - 1)
    {
        currentQualityLevel++;
        ApplyQualitySettings();
        Debug.Log($"Quality increased to level: {currentQualityLevel}");
    }
}

void CheckCulturalContext()
{
    // Update cultural context
    bool newPrayerTime = IsPrayerTime();

    if (newPrayerTime != isPrayerTime)
    {
        isPrayerTime = newPrayerTime;
        OnPrayerTimeChanged(isPrayerTime);
    }
```

```
}

void OnPrayerTimeChanged(bool isPrayer)
{
    if (respectPrayerTimes)
    {
        if (isPrayer)
        {
            // Reduce quality during prayer time for respect
            ApplyPrayerQualitySettings();
        }
        else
        {
            // Restore normal quality
            ApplyQualitySettings();
        }
    }
}

public void SetQualityLevel(int level)
{
    currentQualityLevel = Mathf.Clamp(level, 0, qualityProfiles.Length - 1);
    ApplyQualitySettings();
}

public void ApplyQualitySettings()
{
    if (currentQualityLevel >= qualityProfiles.Length) return;

    QualityLevel profile = qualityProfiles[currentQualityLevel];
```

```
        // Apply Unity quality settings
        UnityEngine.QualitySettings.SetQualityLevel(currentQualityLevel, true);

        // Apply custom quality settings
        ApplyCustomQualitySettings(profile);

        // Apply cultural quality settings
        ApplyCulturalQualitySettings(profile);

        Debug.Log($"Applied quality settings: {profile.name}");
    }

    void ApplyCustomQualitySettings(QualityLevel profile)
    {
        // Texture resolution
        if (profile.textureResolution > 0)
        {
            UnityEngine.QualitySettings.masterTextureLimit = Mathf.Clamp(3 -
profile.textureQuality, 0, 3);
        }

        // Shadow settings
        UnityEngine.QualitySettings.shadows = profile.enableShadows ?
ShadowQuality.All : ShadowQuality.Disable;
        UnityEngine.QualitySettings.shadowResolution =
(ShadowResolution)profile.shadowResolution;

        // Anti-aliasing
        UnityEngine.QualitySettings.antiAliasing = profile.enableAntiAliasing ?
profile.antiAliasingLevel : 0;
```

```
        // VSync
        UnityEngine.QualitySettings.vSyncCount = profile.enableVSync ? 1 : 0;

        // LOD settings
        UnityEngine.QualitySettings.maximumLODLevel = profile.maxLODLevel;
        UnityEngine.QualitySettings.lodBias = profile.lodBias;
    }

    void ApplyCulturalQualitySettings(QualityLevel profile)
    {
        if (!preserveCulturalFidelity) return;

        // Preserve cultural details regardless of quality level
        if (IsReligiousContext() || IsCulturalEvent())
        {
            // Ensure minimum quality for cultural content
            UnityEngine.QualitySettings.masterTextureLimit =
Mathf.Min(UnityEngine.QualitySettings.masterTextureLimit, 1);
            UnityEngine.QualitySettings.lodBias =
Mathf.Max(UnityEngine.QualitySettings.lodBias, 0.8f);

            // Preserve shadow quality for cultural sites
            if (IsReligiousContext())
            {
                UnityEngine.QualitySettings.shadows = ShadowQuality.All;
                UnityEngine.QualitySettings.shadowResolution =
ShadowResolution.High;
            }
        }
    }
```

```
    void ApplyPrayerQualitySettings()
    {
        // Apply special quality settings during prayer time
        QualityLevel prayerProfile = qualityProfiles[currentQualityLevel];

        // Reduce visual intensity for respect
        UnityEngine.QualitySettings.lodBias = prayerProfile.lodBias *
prayerTimeQualityMultiplier;

        // Reduce particle effects
        if (prayerProfile.maxParticleCount > 100)
        {
            // Would limit particle systems
        }

        // Simplify post-processing
        // Would disable intense visual effects
    }

    public QualityLevel GetCurrentQualityProfile()
    {
        return qualityProfiles[currentQualityLevel];
    }

    public void SetCustomQualitySetting(string settingName, object value)
    {
        // Allow runtime modification of quality settings
        QualityLevel customProfile = qualityProfiles[4]; // Custom profile

        switch (settingName)
        {
```

```
            case "textureResolution":
                customProfile.textureResolution = (int)value;
                break;
            case "shadowResolution":
                customProfile.shadowResolution = (int)value;
                break;
            case "enableShadows":
                customProfile.enableShadows = (bool)value;
                break;
            case "enableAntiAliasing":
                customProfile.enableAntiAliasing = (bool)value;
                break;
            case "maxParticleCount":
                customProfile.maxParticleCount = (int)value;
                break;
        }

        if (currentQualityLevel == 4)
        {
            ApplyQualitySettings();
        }
    }

    public void EnableRespectfulMode(bool enabled)
    {
        respectPrayerTimes = enabled;
        preserveCulturalFidelity = enabled;
        maintainTraditionalDetails = enabled;
    }

    float GetPrayerTimeWeight()
```

```
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0f;
}

bool IsPrayerTime()
{
    return GetPrayerTimeWeight() > 0.7f;
}

bool IsReligiousContext()
{
    return GetPrayerTimeWeight() > 0.5f;
}

bool IsCulturalEvent()
{
    // Simplified check
    return Time.time % 86400 > 43200 && Time.time % 86400 < 46800;
}

public QualitySnapshot GetQualitySnapshot()
{
    return new QualitySnapshot
    {
        current_level = currentQualityLevel,
        current_fps = currentFPS,
        average_frame_time = averageFrameTime,
```

```
            target_fps = targetFPS,
            auto_adjust = autoAdjustQuality,
            prayer_time_active = IsPrayerTime(),
            cultural_fidelity = preserveCulturalFidelity
        };
    }

    [System.Serializable]
    public class QualitySnapshot
    {
        public int current_level;
        public float current_fps;
        public float average_frame_time;
        public float target_fps;
        public bool auto_adjust;
        public bool prayer_time_active;
        public bool cultural_fidelity;
    }
}
```

55. BuildConfiguration.cs

```
using UnityEngine;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using System.Collections.Generic;

[BurstCompile]
public class BuildConfiguration : GameSystem
```

```
{
    [BurstCompile]
    struct PlatformOptimization : IJobParallelFor
    {
        [ReadOnly] public NativeArray<float> platformCapabilities;
        [ReadOnly] public NativeArray<bool> featureRequirements;
        [ReadOnly] public NativeArray<int> featurePriorities;
        [WriteOnly] public NativeArray<bool> shouldEnableFeature;
        [WriteOnly] public NativeArray<float> optimizationScores;
        [WriteOnly] public NativeArray<int> platformRecommendations;

        public float memoryLimit;
        public float cpuThreshold;
        public float gpuThreshold;
        public int targetPlatform;

        public void Execute(int index)
        {
            float capability = platformCapabilities[index];
            bool required = featureRequirements[index];
            int priority = featurePriorities[index];

            bool shouldEnable = ShouldEnablePlatformFeature(capability, required,
priority);
            float score = CalculateOptimizationScore(capability, priority);
            int recommendation = GeneratePlatformRecommendation(capability,
required, priority);

            shouldEnableFeature[index] = shouldEnable;
            optimizationScores[index] = score;
            platformRecommendations[index] = recommendation;
```

```
        }

        [BurstCompile]
        bool ShouldEnablePlatformFeature(float capability, bool required, int
priority)
        {
            bool capabilityOk = capability > 0.5f; // 50% threshold
            bool priorityOk = priority > 5 || required;

            return capabilityOk && priorityOk;
        }

        [BurstCompile]
        float CalculateOptimizationScore(float capability, int priority)
        {
            float capabilityScore = capability;
            float priorityScore = priority / 10f;

            return (capabilityScore + priorityScore) * 0.5f;
        }

        [BurstCompile]
        int GeneratePlatformRecommendation(float capability, bool required, int
priority)
        {
            if (required && capability < 0.3f) return 3; // Critical issue
            if (capability < 0.5f) return 2; // Major optimization needed
            if (capability < 0.7f) return 1; // Minor optimization
            return 0; // No optimization needed
        }
    }
```

```
[BurstCompile]
struct MaldivianCulturalBuild : IJob
{
    public NativeArray<bool> prayerTimeOptimizations;
    public NativeArray<bool> culturalContentPackaging;
    public NativeArray<bool> traditionalAssetBundling;
    public NativeArray<bool> communityFeatureFlags;
    public NativeArray<bool> religiousContentHandling;

    public NativeArray<float> culturalSensitivityScores;
    public NativeArray<bool> isBuildConfigurationRespectful;
    public NativeArray<int> appropriateBuildSettings;

    public float currentPrayerTimeWeight;
    public bool isReligiousContext;
    public bool isCommunityBuild;
    public int currentPlatform;
    public string targetLocale;

    public void Execute()
    {
        ValidatePrayerTimeOptimizations();
        ValidateCulturalContentPackaging();
        ValidateTraditionalAssetBundling();
        CalculateBuildConfigurationRespectfulness();
        DetermineAppropriateBuildSettings();
    }

    [BurstCompile]
    void ValidatePrayerTimeOptimizations()
```

```
{
    for (int i = 0; i < prayerTimeOptimizations.Length; i++)
    {
        bool shouldOptimize = currentPrayerTimeWeight > 0.5f;
        prayerTimeOptimizations[i] = shouldOptimize;

        if (shouldOptimize)
        {
            culturalSensitivityScores[i] = 0.95f;
        }
    }
}

[BurstCompile]
void ValidateCulturalContentPackaging()
{
    for (int i = 0; i < culturalContentPackaging.Length; i++)
    {
        bool shouldPackage = isReligiousContext || isCommunityBuild;
        culturalContentPackaging[i] = shouldPackage;

        if (shouldPackage)
        {
            isBuildConfigurationRespectful[i] = true;
        }
    }
}

[BurstCompile]
void ValidateTraditionalAssetBundling()
{
```

```
        for (int i = 0; i < traditionalAssetBundling.Length; i++)
        {
            bool isTraditional = targetLocale == "dv" || targetLocale == "en";
            traditionalAssetBundling[i] = isTraditional;

            if (isTraditional)
            {
                culturalSensitivityScores[prayerTimeOptimizations.Length + i] =
0.90f;
            }
        }
    }

    [BurstCompile]
    void CalculateBuildConfigurationRespectfulness()
    {
        float totalScore = 0.0f;
        int totalChecks = 0;

        for (int i = 0; i < prayerTimeOptimizations.Length; i++)
        {
            if (prayerTimeOptimizations[i]) totalScore += 1.0f;
            totalChecks++;
        }

        for (int i = 0; i < culturalContentPackaging.Length; i++)
        {
            if (culturalContentPackaging[i]) totalScore += 1.0f;
            totalChecks++;
        }
```

```
        float averageScore = totalChecks > 0 ? totalScore / totalChecks : 0.0f;

        for (int i = 0; i < isBuildConfigurationRespectful.Length; i++)
        {
            isBuildConfigurationRespectful[i] = averageScore > 0.8f;
        }
    }

    [BurstCompile]
    void DetermineAppropriateBuildSettings()
    {
        int setting = 1; // Standard build

        if (currentPrayerTimeWeight > 0.7f) setting = 2; // Optimized build
        if (isReligiousContext) setting = 3; // Respectful build
        if (isCommunityBuild) setting = 4; // Community build

        for (int i = 0; i < appropriateBuildSettings.Length; i++)
        {
            appropriateBuildSettings[i] = setting;
        }
    }
}

public static BuildConfiguration Instance { get; private set; }

[Header("Platform Configuration")]
public RuntimePlatform targetPlatform = RuntimePlatform.Android;
public bool autoDetectPlatform = true;
public bool enablePlatformOptimizations = true;
public bool enableDebugBuild = false;
```

```
[Header("Cultural Build Settings")]
public bool respectPrayerTimes = true;
public bool packageCulturalContent = true;
public bool maintainTraditionalAssets = true;
public bool enableRespectfulDistribution = true;

[Header("Mobile Build Settings")]
public bool optimizeForMobile = true;
public bool stripUnusedAssets = true;
public bool compressTextures = true;
public bool useAssetBundles = true;
public int maxBundleSize = 100; // MB

[Header("Build Features")]
public bool enableBurstCompilation = true;
public bool enableMathematics = true;
public bool enableCollections = true;
public bool enableJobsSystem = true;

private BuildPlatform currentBuildPlatform;
private Dictionary<string, BuildAsset> buildAssets;
private List<string> culturalAssetPaths;
private bool isPrayerTime;
private string buildVersion;
private int buildNumber;

public enum BuildPlatform
{
    Android = 0,
    iOS = 1,
```

```
    Windows = 2,
    MacOS = 3,
    WebGL = 4,
    Linux = 5
}

[System.Serializable]
public class BuildAsset
{
    public string assetPath;
    public bool isEssential;
    public bool isCultural;
    public int platformPriority;
    public float estimatedSize;
    public string culturalContext;
    public bool includeInBuild;
}

void Awake()
{
    if (Instance == null)
    {
        Instance = this;
        DontDestroyOnLoad(gameObject);
    }
    else
    {
        Destroy(gameObject);
    }
}
```

```
    public override void Initialize(MainGameManager manager)
    {
        base.Initialize(manager);
        InitializeBuildConfiguration();
    }

    void InitializeBuildConfiguration()
    {
        // Initialize collections
        buildAssets = new Dictionary<string, BuildAsset>();
        culturalAssetPaths = new List<string>();

        // Detect current platform
        if (autoDetectPlatform)
        {
            DetectPlatform();
        }

        // Initialize platform optimization
        int featureCount = 25;
        var capabilities = new NativeArray<float>(featureCount, Allocator.TempJob);
        var requirements = new NativeArray<bool>(featureCount,
Allocator.TempJob);
        var priorities = new NativeArray<int>(featureCount, Allocator.TempJob);
        var shouldEnable = new NativeArray<bool>(featureCount,
Allocator.TempJob);
        var scores = new NativeArray<float>(featureCount, Allocator.TempJob);
        var recommendations = new NativeArray<int>(featureCount,
Allocator.TempJob);

        // Initialize with sample platform data
```

```
for (int i = 0; i < featureCount; i++)
{
    capabilities[i] = UnityEngine.Random.Range(0.2f, 1.0f);
    requirements[i] = UnityEngine.Random.Range(0, 2) == 1;
    priorities[i] = UnityEngine.Random.Range(1, 10);
}

var optimizationJob = new PlatformOptimization
{
    platformCapabilities = capabilities,
    featureRequirements = requirements,
    featurePriorities = priorities,
    shouldEnableFeature = shouldEnable,
    optimizationScores = scores,
    platformRecommendations = recommendations,
    memoryLimit = SystemInfo.systemMemorySize,
    cpuThreshold = 2.0f, // 2 GHz baseline
    gpuThreshold = 1000f, // GPU performance baseline
    targetPlatform = (int)currentBuildPlatform
};

// Initialize cultural build validation
var prayerOptimizations = new NativeArray<bool>(8, Allocator.TempJob);
var culturalPackaging = new NativeArray<bool>(10, Allocator.TempJob);
var traditionalBundling = new NativeArray<bool>(12, Allocator.TempJob);
var communityFlags = new NativeArray<bool>(6, Allocator.TempJob);
var religiousHandling = new NativeArray<bool>(4, Allocator.TempJob);
var sensitivityScores = new NativeArray<float>(60, Allocator.TempJob);
var buildRespectful = new NativeArray<bool>(60, Allocator.TempJob);
var buildSettings = new NativeArray<int>(60, Allocator.TempJob);
```

```
        var culturalJob = new MaldivianCulturalBuild
        {
            prayerTimeOptimizations = prayerOptimizations,
            culturalContentPackaging = culturalPackaging,
            traditionalAssetBundling = traditionalBundling,
            communityFeatureFlags = communityFlags,
            religiousContentHandling = religiousHandling,
            culturalSensitivityScores = sensitivityScores,
            isBuildConfigurationRespectful = buildRespectful,
            appropriateBuildSettings = buildSettings,
            currentPrayerTimeWeight = GetPrayerTimeWeight(),
            isReligiousContext = IsReligiousContext(),
            isCommunityBuild = IsCommunityBuild(),
            currentPlatform = (int)currentBuildPlatform,
            targetLocale = "dv" // Dhivehi as primary cultural locale
        };

        JobHandle optimizationHandle = optimizationJob.Schedule(featureCount,
8);
        JobHandle culturalHandle = culturalJob.Schedule(optimizationHandle);
        culturalHandle.Complete();

        // Process results
        ProcessOptimizationResults(shouldEnable, scores, recommendations);
        ProcessCulturalValidation(prayerOptimizations, culturalPackaging,
sensitivityScores, buildSettings);

        // Cleanup
        capabilities.Dispose();
        requirements.Dispose();
        priorities.Dispose();
```

```
        shouldEnable.Dispose();
        scores.Dispose();
        recommendations.Dispose();
        prayerOptimizations.Dispose();
        culturalPackaging.Dispose();
        traditionalBundling.Dispose();
        communityFlags.Dispose();
        religiousHandling.Dispose();
        sensitivityScores.Dispose();
        buildRespectful.Dispose();
        buildSettings.Dispose();

        RegisterBuildAssets();
        StartBuildMonitoring();
    }

    void ProcessOptimizationResults(NativeArray<bool> shouldEnable,
NativeArray<float> scores, NativeArray<int> recommendations)
    {
        int enableCount = 0;
        float avgScore = 0f;

        for (int i = 0; i < shouldEnable.Length; i++)
        {
            if (shouldEnable[i]) enableCount++;
            avgScore += scores[i];
        }

        avgScore /= shouldEnable.Length;
```

```
        Debug.Log($"Platform optimization: {enableCount} features should be enabled, average score: {avgScore:F2}");
    }

    void ProcessCulturalValidation(NativeArray<bool> prayer, NativeArray<bool> cultural, NativeArray<float> sensitivity, NativeArray<int> settings)
    {
        int optimizationCount = 0;
        for (int i = 0; i < prayer.Length; i++)
        {
            if (prayer[i]) optimizationCount++;
        }

        int packagingCount = 0;
        for (int i = 0; i < cultural.Length; i++)
        {
            if (cultural[i]) packagingCount++;
        }

        Debug.Log($"Cultural validation: {optimizationCount} prayer optimizations, {packagingCount} cultural packagings");
    }

    void StartBuildMonitoring()
    {
        InvokeRepeating("MonitorBuildPerformance", 1f, 5f);
        InvokeRepeating("CheckCulturalContext", 2f, 10f);
        InvokeRepeating("ValidateBuildConfiguration", 10f, 30f);
    }

    void DetectPlatform()
```

```
{
    currentBuildPlatform = Application.platform switch
    {
        RuntimePlatform.Android => BuildPlatform.Android,
        RuntimePlatform.IPhonePlayer => BuildPlatform.iOS,
        RuntimePlatform.WindowsPlayer => BuildPlatform.Windows,
        RuntimePlatform.OSXPlayer => BuildPlatform.MacOS,
        RuntimePlatform.WebGLPlayer => BuildPlatform.WebGL,
        RuntimePlatform.LinuxPlayer => BuildPlatform.Linux,
        _ => BuildPlatform.Android
    };

    Debug.Log($"Detected platform: {currentBuildPlatform}");
}

public void ConfigureForPlatform(BuildPlatform platform)
{
    currentBuildPlatform = platform;

    switch (platform)
    {
        case BuildPlatform.Android:
            ConfigureForAndroid();
            break;
        case BuildPlatform.iOS:
            ConfigureForiOS();
            break;
        case BuildPlatform.Windows:
            ConfigureForWindows();
            break;
        case BuildPlatform.MacOS:
```

```
            ConfigureForMacOS();
            break;
        case BuildPlatform.WebGL:
            ConfigureForWebGL();
            break;
        case BuildPlatform.Linux:
            ConfigureForLinux();
            break;
    }
}

void ConfigureForAndroid()
{
    // Android-specific optimizations
    optimizeForMobile = true;
    compressTextures = true;
    useAssetBundles = true;
    maxBundleSize = 50; // 50 MB for mobile
    stripUnusedAssets = true;

    // Unity settings
    UnityEngine.QualitySettings.SetQualityLevel(1, true); // Medium quality
    UnityEngine.Application.targetFrameRate = 60;
}

void ConfigureForiOS()
{
    // iOS-specific optimizations
    optimizeForMobile = true;
    compressTextures = true;
    useAssetBundles = true;
```

```
    maxBundleSize = 100; // 100 MB for iOS
    stripUnusedAssets = true;

    // Unity settings
    UnityEngine.QualitySettings.SetQualityLevel(2, true); // High quality for iOS
    UnityEngine.Application.targetFrameRate = 60;
}

void ConfigureForWindows()
{
    // Windows-specific optimizations
    optimizeForMobile = false;
    compressTextures = false; // Less compression for desktop
    useAssetBundles = false;
    maxBundleSize = 500; // 500 MB for desktop
    stripUnusedAssets = false;

    // Unity settings
    UnityEngine.QualitySettings.SetQualityLevel(3, true); // Ultra quality
    UnityEngine.Application.targetFrameRate = -1; // Uncapped
}

void ConfigureForMacOS()
{
    // MacOS-specific optimizations
    ConfigureForWindows(); // Similar to Windows
}

void ConfigureForWebGL()
{
    // WebGL-specific optimizations
```

```
        optimizeForMobile = true;
        compressTextures = true;
        useAssetBundles = true;
        maxBundleSize = 25; // 25 MB for WebGL
        stripUnusedAssets = true;

        // Unity settings
        UnityEngine.QualitySettings.SetQualityLevel(1, true); // Medium quality
        UnityEngine.Application.targetFrameRate = 30;
    }

    void ConfigureForLinux()
    {
        // Linux-specific optimizations
        ConfigureForWindows(); // Similar to Windows
    }

    void RegisterBuildAssets()
    {
        // Register essential build assets
        RegisterAsset("Textures/UI/MainMenu", AssetType.Texture, true, 2f);
        RegisterAsset("Textures/UI/HUD", AssetType.Texture, true, 3f);
        RegisterAsset("Models/Player/Character", AssetType.Model, true, 5f);
        RegisterAsset("Audio/Music/MainTheme", AssetType.Audio, true, 4f);

        // Register cultural assets
        RegisterAsset("Textures/Cultural/Mosque", AssetType.Texture, false, 2f,
"Religious");
        RegisterAsset("Models/Cultural/TraditionalBoat", AssetType.Model, false,
6f, "Traditional");
```

```
        RegisterAsset("Audio/Cultural/Boduberu", AssetType.Audio, false, 3f,
"Cultural");

        // Register platform-specific assets
        if (optimizeForMobile)
        {
            RegisterAsset("Textures/Mobile/LowRes", AssetType.Texture, true, 1f);
            RegisterAsset("Models/Mobile/Simplified", AssetType.Model, true, 3f);
        }
    }

    void RegisterAsset(string path, AssetType type, bool essential, float size,
string culturalContext = "General")
    {
        BuildAsset asset = new BuildAsset
        {
            assetPath = path,
            isEssential = essential,
            isCultural = !string.IsNullOrEmpty(culturalContext) && culturalContext !=
"General",
            platformPriority = GetPlatformPriority(type),
            estimatedSize = size,
            culturalContext = culturalContext,
            includeInBuild = essential || !optimizeForMobile // Include all essential
assets, others based on mobile optimization
        };

        buildAssets[path] = asset;

        if (asset.isCultural)
        {
```

```
            culturalAssetPaths.Add(path);
        }
    }

    int GetPlatformPriority(AssetType type)
    {
        return type switch
        {
            AssetType.Texture => 8,
            AssetType.Model => 7,
            AssetType.Audio => 6,
            AssetType.Prefab => 9,
            AssetType.Material => 5,
            AssetType.Cultural => 10,
            AssetType.Religious => 10,
            AssetType.Traditional => 9,
            _ => 5
        };
    }

    public void BuildAssetBundles()
    {
        if (!useAssetBundles) return;

        // Organize assets into bundles
        Dictionary<string, List<string>> bundles = new Dictionary<string,
List<string>>();

        // Create cultural asset bundle
        if (culturalAssetPaths.Count > 0)
        {
```

```
            bundles["cultural_assets"] = new List<string>(culturalAssetPaths);
        }

        // Create platform-specific bundles
        bundles["essential_assets"] = new List<string>();
        bundles["platform_specific"] = new List<string>();

        foreach (var kvp in buildAssets)
        {
            if (kvp.Value.includeInBuild)
            {
                if (kvp.Value.isEssential)
                {
                    bundles["essential_assets"].Add(kvp.Key);
                }
                else
                {
                    bundles["platform_specific"].Add(kvp.Key);
                }
            }
        }

        // Simulate bundle creation (in production, would use Unity's AssetBundle
system)
        Debug.Log($"Created {bundles.Count} asset bundles");

        foreach (var bundle in bundles)
        {
            Debug.Log($"Bundle: {bundle.Key} contains {bundle.Value.Count}
assets");
        }
```

```
}

public void SetBuildVersion(string version, int buildNum)
{
    buildVersion = version;
    buildNumber = buildNum;

    PlayerSettings.bundleVersion = version;
    PlayerSettings.Android.bundleVersionCode = buildNum;
}

public void EnableCulturalBuildMode(bool enabled)
{
    packageCulturalContent = enabled;
    maintainTraditionalAssets = enabled;
    enableRespectfulDistribution = enabled;

    // Reconfigure build settings
    ConfigureForPlatform(currentBuildPlatform);
}

void MonitorBuildPerformance()
{
    // Monitor build performance and size
    float totalSize = 0f;
    int includedAssets = 0;

    foreach (var asset in buildAssets.Values)
    {
        if (asset.includeInBuild)
        {
```

```
            totalSize += asset.estimatedSize;
            includedAssets++;
        }
    }

    Debug.Log($"Build: {includedAssets} assets, {totalSize:F1} MB total");

    // Check if build size is within limits
    float maxSize = currentBuildPlatform switch
    {
        BuildPlatform.Android => 100f, // 100 MB for mobile
        BuildPlatform.iOS => 200f, // 200 MB for iOS
        BuildPlatform.WebGL => 50f, // 50 MB for WebGL
        _ => 500f // 500 MB for desktop
    };

    if (totalSize > maxSize)
    {
        Debug.LogWarning($"Build size exceeds platform limit: {totalSize:F1} MB
> {maxSize} MB");
        OptimizeBuildSize();
    }
}

void OptimizeBuildSize()
{
    // Reduce build size by excluding non-essential assets
    foreach (var kvp in buildAssets)
    {
        if (!kvp.Value.isEssential && kvp.Value.estimatedSize > 5f)
        {
```

```
            kvp.Value.includeInBuild = false;
        }
    }
}

void CheckCulturalContext()
{
    // Update cultural context
    bool newPrayerTime = IsPrayerTime();

    if (newPrayerTime != isPrayerTime)
    {
        isPrayerTime = newPrayerTime;
        OnPrayerTimeChanged(isPrayerTime);
    }
}

void OnPrayerTimeChanged(bool isPrayer)
{
    if (respectPrayerTimes)
    {
        // Adjust build configuration for prayer time
        // Could affect asset loading priorities, etc.
    }
}

void ValidateBuildConfiguration()
{
    // Validate that build configuration is culturally respectful
    if (enableRespectfulDistribution)
    {
```

```
        bool isRespectfulTime = !IsPrayerTime();

        if (!isRespectfulTime)
        {
            // Defer non-essential build operations
            Debug.Log("Build operations deferred for cultural respect");
        }
    }
}

bool IsCommunityBuild()
{
    // Check if this is a community-focused build
    return culturalAssetPaths.Count > 10; // Simplified check
}

float GetPrayerTimeWeight()
{
    if (PrayerTimeSystem.Instance != null)
    {
        return PrayerTimeSystem.Instance.GetNextPrayerProximity();
    }
    return 0f;
}

bool IsPrayerTime()
{
    return GetPrayerTimeWeight() > 0.7f;
}

bool IsReligiousContext()
```

```
{
    return GetPrayerTimeWeight() > 0.5f;
}

public BuildSnapshot GetBuildSnapshot()
{
    int includedCount = 0;
    float totalSize = 0f;

    foreach (var asset in buildAssets.Values)
    {
        if (asset.includeInBuild)
        {
            includedCount++;
            totalSize += asset.estimatedSize;
        }
    }

    return new BuildSnapshot
    {
        target_platform = currentBuildPlatform.ToString(),
        build_version = buildVersion,
        build_number = buildNumber,
        total_assets = buildAssets.Count,
        included_assets = includedCount,
        estimated_size_mb = totalSize,
        cultural_assets = culturalAssetPaths.Count,
        prayer_time_respect = respectPrayerTimes
    };
}
```

```
    [System.Serializable]
    public class BuildSnapshot
    {
        public string target_platform;
        public string build_version;
        public int build_number;
        public int total_assets;
        public int included_assets;
        public float estimated_size_mb;
        public int cultural_assets;
        public bool prayer_time_respect;
    }
}
```

56. ProjectSettings.asset

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!129 &1
PlayerSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 24
  productGUID: b0bc8f72-a9d8-4c5e-9f3a-7d2e9f8c5a1b
  AndroidProfiler: 0
  AndroidFilterTouchesWhenObscured: 0
  AndroidEnableSustainedPerformanceMode: 1
  defaultScreenOrientation: 3
```

```
  targetDevice: 2
  useOnDemandResources: 0
  accelerometerFrequency: 60
  companyName: RAAJJE VAGU STUDIOS
  productName: RAAJJE VAGU AUTO - THE ALBAKO CHRONICLES
  defaultCursor: {fileID: 0}
  cursorHotspot: {x: 0, y: 0}
  m_SplashScreenBackgroundColor: {r: 0.13725491, g: 0.12156863, b:
0.1254902, a: 1}
  m_ShowUnitySplashScreen: 1
  m_ShowUnitySplashLogo: 1
  m_SplashScreenOverlayOpacity: 1
  m_SplashScreenAnimation: 1
  m_SplashScreenLogoStyle: 1
  m_SplashScreenDrawMode: 0
  m_SplashScreenBackgroundAnimationZoom: 1
  m_SplashScreenLogoAnimationZoom: 1
  m_SplashScreenBackgroundLandscapeAspectRatio: 2
  m_SplashScreenBackgroundPortraitAspectRatio: 1
  m_SplashScreenLogos: []
  m_VirtualRealitySDKs: []
  m_TargetPixelDensity: 30
  m_ResolutionScalingMode: 0
  androidSupportedAspectRatio: 1
  androidMaxAspectRatio: 2.1
  applicationIdentifier:
    Android: com.raajjevaguauto.albakochronicles
    iPhone: com.raajjevaguauto.albakochronicles
  buildNumber:
    Standalone: 0
    iPhone: 0
```

tvOS: 0
AndroidBundleVersionCode: 1
AndroidMinSdkVersion: 21
AndroidTargetSdkVersion: 33
AndroidPreferredInstallLocation: 1
aotOptions:
stripEngineCode: 1
iPhoneStrippingLevel: 0
iPhoneScriptCallOptimization: 0
ForceInternetPermission: 0
ForceSDCardPermission: 0
CreateWallpaper: 0
APKExpansionFiles: 0
keepLoadedShadersAlive: 0
StripUnusedMeshComponents: 1
VertexCompressionCanChange: 0
m_MTRendering: 1
m_MTRenderingCanChange: 0
iosShowActivityIndicatorOnLoading: -1
androidShowActivityIndicatorOnLoading: -1
iosUseCustomAppBackgroundBehavior: 0
iosAllowHTTPDownload: 1
allowedAutorotateToPortrait: 1
allowedAutorotateToPortraitUpsideDown: 1
allowedAutorotateToLandscapeRight: 1
allowedAutorotateToLandscapeLeft: 1
useOSAutorotation: 1
use32BitDisplayBuffer: 1
preserveFramebufferAlpha: 0
disableDepthAndStencilBuffers: 0
androidStartInFullscreen: 1

androidRenderOutsideSafeArea: 1
androidUseSwappy: 1
androidBlitType: 0
defaultIsNativeResolution: 1
macRetinaSupport: 1
runInBackground: 0
captureSingleScreen: 0
muteOtherAudioSources: 0
Prepare IOS For Recording: 0
Force IOS Speakers When Recording: 0
deferSystemGesturesMode: 0
hideHomeButton: 0
submitAnalytics: 1
usePlayerLog: 1
bakeCollisionMeshes: 0
forceSingleInstance: 0
useFlipModelSwapchain: 1
resizableWindow: 0
useMacAppStoreValidation: 0
macAppStoreCategory: public.app-category.games
gpuSkinning: 1
xboxPIXTextureCapture: 0
xboxEnableAvatar: 0
xboxEnableKinect: 0
xboxEnableKinectAutoTracking: 0
xboxEnableFitness: 0
visibleInBackground: 1
allowFullscreenSwitch: 1
graphicsJobMode: 0
fullscreenMode: 1
xboxSpeechDB: 0

xboxEnableHeadOrientation: 0
xboxEnableGuest: 0
xboxEnablePIXSampling: 0
metalFramebufferOnly: 0
xboxOneResolution: 0
xboxOneSResolution: 0
xboxOneXResolution: 3
xboxOneMonoLoggingLevel: 0
xboxOneLoggingLevel: 1
xboxOneDisableEsram: 0
xboxOneEnableTypeOptimization: 0
xboxOnePresentImmediateThreshold: 0
switchQueueCommandMemory: 0
switchQueueControlMemory: 16384
switchQueueComputeMemory: 262144
switchNVNShaderPoolsGranularity: 33554432
switchNVNDefaultPoolsGranularity: 16777216
switchNVNOtherPoolsGranularity: 33554432
switchNVNMaxPublicTextureIDCount: 0
switchNVNMaxPublicSamplerIDCount: 0
stadiaPresentMode: 0
stadiaTargetFramerate: 0
vulkanNumSwapchainBuffers: 3
vulkanEnableSetSRGBWrite: 0
vulkanEnablePreTransform: 0
vulkanEnableLateAcquireNextImage: 0
vulkanEnableCommandBufferRecycling: 1
m_SupportedAspectRatios:
  4:3: 1
  5:4: 1
  16:10: 1

```
  16:9: 1
  Others: 1
bundleVersion: 1.0.0
preloadedAssets: []
metroInputSource: 0
wsaTransparentSwapchain: 0
m_HolographicPauseOnTrackingLoss: 1
xboxOneDisableKinectGpuReservation: 1
xboxOneEnable7thCore: 1
vrSettings:
  enable360StereoCapture: 0
isWsaHolographicRemotingEnabled: 0
enableFrameTimingStats: 1
enableOpenGLProfilerGPURecorders: 1
useHDRDisplay: 0
D3DHDRBitDepth: 0
m_ColorGamuts: 00000000
targetPixelDensity: 30
resolutionScalingMode: 0
resetResolutionOnWindowResize: 0
androidSupportedAspectRatio: 1
androidMaxAspectRatio: 2.1
applicationIdentifier:
  Android: com.raajjevaguauto.albakochronicles
  iPhone: com.raajjevaguauto.albakochronicles
buildNumber:
  Standalone: 0
  iPhone: 0
  tvOS: 0
AndroidBundleVersionCode: 1
AndroidMinSdkVersion: 21
```

AndroidTargetSdkVersion: 33
AndroidPreferredInstallLocation: 1
aotOptions:
stripEngineCode: 1
iPhoneStrippingLevel: 0
iPhoneScriptCallOptimization: 0
ForceInternetPermission: 0
ForceSDCardPermission: 0
CreateWallpaper: 0
APKExpansionFiles: 0
keepLoadedShadersAlive: 0
StripUnusedMeshComponents: 1
VertexCompressionCanChange: 0
m_MTRendering: 1
m_MTRenderingCanChange: 0
iosShowActivityIndicatorOnLoading: -1
androidShowActivityIndicatorOnLoading: -1
iosUseCustomAppBackgroundBehavior: 0
iosAllowHTTPDownload: 1
allowedAutorotateToPortrait: 1
allowedAutorotateToPortraitUpsideDown: 1
allowedAutorotateToLandscapeRight: 1
allowedAutorotateToLandscapeLeft: 1
useOSAutorotation: 1
use32BitDisplayBuffer: 1
preserveFramebufferAlpha: 0
disableDepthAndStencilBuffers: 0
androidStartInFullscreen: 1
androidRenderOutsideSafeArea: 1
androidUseSwappy: 1
androidBlitType: 0

defaultIsNativeResolution: 1
macRetinaSupport: 1
runInBackground: 0
captureSingleScreen: 0
muteOtherAudioSources: 0
Prepare IOS For Recording: 0
Force IOS Speakers When Recording: 0
deferSystemGesturesMode: 0
hideHomeButton: 0
submitAnalytics: 1
usePlayerLog: 1
bakeCollisionMeshes: 0
forceSingleInstance: 0
useFlipModelSwapchain: 1
resizableWindow: 0
useMacAppStoreValidation: 0
macAppStoreCategory: public.app-category.games
gpuSkinning: 1
xboxPIXTextureCapture: 0
xboxEnableAvatar: 0
xboxEnableKinect: 0
xboxEnableKinectAutoTracking: 0
xboxEnableFitness: 0
visibleInBackground: 1
allowFullscreenSwitch: 1
graphicsJobMode: 0
fullscreenMode: 1
xboxSpeechDB: 0
xboxEnableHeadOrientation: 0
xboxEnableGuest: 0
xboxEnablePIXSampling: 0

metalFramebufferOnly: 0
xboxOneResolution: 0
xboxOneSResolution: 0
xboxOneXResolution: 3
xboxOneMonoLoggingLevel: 0
xboxOneLoggingLevel: 1
xboxOneDisableEsram: 0
xboxOneEnableTypeOptimization: 0
xboxOnePresentImmediateThreshold: 0
switchQueueCommandMemory: 0
switchQueueControlMemory: 16384
switchQueueComputeMemory: 262144
switchNVNShaderPoolsGranularity: 33554432
switchNVNDefaultPoolsGranularity: 16777216
switchNVNOtherPoolsGranularity: 33554432
switchNVNMaxPublicTextureIDCount: 0
switchNVNMaxPublicSamplerIDCount: 0
stadiaPresentMode: 0
stadiaTargetFramerate: 0
vulkanNumSwapchainBuffers: 3
vulkanEnableSetSRGBWrite: 0
vulkanEnablePreTransform: 0
vulkanEnableLateAcquireNextImage: 0
vulkanEnableCommandBufferRecycling: 1
m_SupportedAspectRatios:
  4:3: 1
  5:4: 1
  16:10: 1
  16:9: 1
  Others: 1
bundleVersion: 1.0.0

```
  preloadedAssets: []
  metroInputSource: 0
  wsaTransparentSwapchain: 0
  m_HolographicPauseOnTrackingLoss: 1
  xboxOneDisableKinectGpuReservation: 1
  xboxOneEnable7thCore: 1
  vrSettings:
    enable360StereoCapture: 0
  isWsaHolographicRemotingEnabled: 0
  enableFrameTimingStats: 1
  enableOpenGLProfilerGPURecorders: 1
  useHDRDisplay: 0
  D3DHDRBitDepth: 0
  m_ColorGamuts: 00000000
  targetPixelDensity: 30
  resolutionScalingMode: 0
  resetResolutionOnWindowResize: 0
```

57. QualitySettings.asset

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!47 &1
QualitySettings:
  m_ObjectHideFlags: 0
  serializedVersion: 5
  m_CurrentQuality: 2
  m_QualitySettings:
```

```
- serializedVersion: 2
  name: Low Quality (Mobile)
  pixelLightCount: 0
  shadows: 0
  shadowResolution: 0
  shadowProjection: 1
  shadowCascades: 1
  shadowDistance: 20
  shadowNearPlaneOffset: 2
  shadowCascade2Split: 0.33333334
  shadowCascade4Split: {x: 0.06666667, y: 0.2, z: 0.46666667}
  shadowmaskMode: 0
  skinWeights: 1
  globalTextureMipmapLimit: 2
  textureMipmapLimitSettings: []
  anisotropicTextures: 0
  antiAliasing: 0
  softParticles: 0
  softVegetation: 0
  realtimeReflectionProbes: 0
  billboardsFaceCameraPosition: 0
  useLegacyDetailDistribution: 1
  vSyncCount: 0
  lodBias: 0.3
  maximumLODLevel: 2
  enableLODCrossFade: 0
  streamingMipmapsActive: 0
  streamingMipmapsMemoryBudget: 512
  streamingMipmapsAddAllCameras: 1
  streamingMipmapsRenderersPerFrame: 512
  streamingMipmapsMaxLevelReduction: 2
```

streamingMipmapsMaxFileIORequests: 1024
particles: 1
shadows: 0
shadowsWorkWithLowSettings: 0
vSync: 0
realtimeGICPUUsage: 25
textureQuality: 0
resolutionScalingFixedDPIFactor: 1
customRenderingSettings: 0
terrainQualityOverrides: 0
terrainPixelError: 1
terrainDetailDensityScale: 1
terrainDetailDensityMultiplier: 1
terrainBasemapDistance: 1000
terrainDetailDistance: 80
terrainTreeDistance: 2000
terrainBillboardStart: 50
terrainFadeLength: 5
terrainMaxTrees: 50
asyncAssetUploadPersistentBuffer: 1
asyncAssetUploadBufferSize: 16
asyncAssetUploadPersistentBuffer: 1
asyncAssetUploadBufferSize: 16
overridePlatformResolution: 0
resolutionScalingMode: 0
androidSupportedAspectRatio: 1
androidMaxAspectRatio: 2.1
androidBlitType: 0
androidStartInFullscreen: 1
androidRenderOutsideSafeArea: 1
androidUseSwappy: 1

```
  androidSupportedAspectRatio: 1
  androidMaxAspectRatio: 2.1
  androidBlitType: 0
  androidStartInFullscreen: 1
  androidRenderOutsideSafeArea: 1
  androidUseSwappy: 1
  m_CulturalSettings:
    m_RespectPrayerTimes: 1
    m_PreserveCulturalFidelity: 0
    m_ReducedVisualIntensity: 1
    m_DisableLoudEffects: 1
    m_SimplifiedUI: 1

- serializedVersion: 2
  name: Medium Quality (Balanced)
  pixelLightCount: 1
  shadows: 1
  shadowResolution: 1
  shadowProjection: 1
  shadowCascades: 2
  shadowDistance: 40
  shadowNearPlaneOffset: 2
  shadowCascade2Split: 0.33333334
  shadowCascade4Split: {x: 0.06666667, y: 0.2, z: 0.46666667}
  shadowmaskMode: 0
  skinWeights: 2
  globalTextureMipmapLimit: 1
  textureMipmapLimitSettings: []
  anisotropicTextures: 1
  antiAliasing: 2
  softParticles: 0
```

softVegetation: 0
realtimeReflectionProbes: 0
billboardsFaceCameraPosition: 0
useLegacyDetailDistribution: 1
vSyncCount: 1
lodBias: 0.7
maximumLODLevel: 1
enableLODCrossFade: 1
streamingMipmapsActive: 0
streamingMipmapsMemoryBudget: 512
streamingMipmapsAddAllCameras: 1
streamingMipmapsRenderersPerFrame: 512
streamingMipmapsMaxLevelReduction: 2
streamingMipmapsMaxFileIORequests: 1024
particles: 1
shadows: 1
shadowsWorkWithLowSettings: 1
vSync: 1
realtimeGICPUUsage: 50
textureQuality: 1
resolutionScalingFixedDPIFactor: 1
customRenderingSettings: 0
terrainQualityOverrides: 0
terrainPixelError: 1
terrainDetailDensityScale: 1
terrainDetailDensityMultiplier: 1
terrainBasemapDistance: 1000
terrainDetailDistance: 120
terrainTreeDistance: 2000
terrainBillboardStart: 50
terrainFadeLength: 5

```
  terrainMaxTrees: 100
  asyncAssetUploadPersistentBuffer: 1
  asyncAssetUploadBufferSize: 16
  overridePlatformResolution: 0
  resolutionScalingMode: 0
  androidSupportedAspectRatio: 1
  androidMaxAspectRatio: 2.1
  androidBlitType: 0
  androidStartInFullscreen: 1
  androidRenderOutsideSafeArea: 1
  androidUseSwappy: 1
  m_CulturalSettings:
    m_RespectPrayerTimes: 1
    m_PreserveCulturalFidelity: 1
    m_ReducedVisualIntensity: 0
    m_DisableLoudEffects: 0
    m_SimplifiedUI: 0

- serializedVersion: 2
  name: High Quality (Cultural)
  pixelLightCount: 2
  shadows: 2
  shadowResolution: 2
  shadowProjection: 1
  shadowCascades: 4
  shadowDistance: 80
  shadowNearPlaneOffset: 2
  shadowCascade2Split: 0.33333334
  shadowCascade4Split: {x: 0.06666667, y: 0.2, z: 0.46666667}
  shadowmaskMode: 1
  skinWeights: 4
```

globalTextureMipmapLimit: 0
textureMipmapLimitSettings: []
anisotropicTextures: 2
antiAliasing: 4
softParticles: 1
softVegetation: 1
realtimeReflectionProbes: 1
billboardsFaceCameraPosition: 1
useLegacyDetailDistribution: 1
vSyncCount: 1
lodBias: 1.0
maximumLODLevel: 0
enableLODCrossFade: 1
streamingMipmapsActive: 0
streamingMipmapsMemoryBudget: 512
streamingMipmapsAddAllCameras: 1
streamingMipmapsRenderersPerFrame: 512
streamingMipmapsMaxLevelReduction: 2
streamingMipmapsMaxFileIORequests: 1024
particles: 1
shadows: 2
shadowsWorkWithLowSettings: 1
vSync: 1
realtimeGICPUUsage: 75
textureQuality: 2
resolutionScalingFixedDPIFactor: 1
customRenderingSettings: 0
terrainQualityOverrides: 0
terrainPixelError: 1
terrainDetailDensityScale: 1
terrainDetailDensityMultiplier: 1

```
    terrainBasemapDistance: 1000
    terrainDetailDistance: 150
    terrainTreeDistance: 2000
    terrainBillboardStart: 50
    terrainFadeLength: 5
    terrainMaxTrees: 200
    asyncAssetUploadPersistentBuffer: 1
    asyncAssetUploadBufferSize: 16
    overridePlatformResolution: 0
    resolutionScalingMode: 0
    androidSupportedAspectRatio: 1
    androidMaxAspectRatio: 2.1
    androidBlitType: 0
    androidStartInFullscreen: 1
    androidRenderOutsideSafeArea: 1
    androidUseSwappy: 1
    m_CulturalSettings:
      m_RespectPrayerTimes: 0
      m_PreserveCulturalFidelity: 1
      m_ReducedVisualIntensity: 0
      m_DisableLoudEffects: 0
      m_SimplifiedUI: 0

  - serializedVersion: 2
    name: Ultra Quality (Premium)
    pixelLightCount: 4
    shadows: 2
    shadowResolution: 3
    shadowProjection: 1
    shadowCascades: 4
    shadowDistance: 150
```

shadowNearPlaneOffset: 2
shadowCascade2Split: 0.33333334
shadowCascade4Split: {x: 0.06666667, y: 0.2, z: 0.46666667}
shadowmaskMode: 1
skinWeights: 255
globalTextureMipmapLimit: 0
textureMipmapLimitSettings: []
anisotropicTextures: 2
antiAliasing: 8
softParticles: 1
softVegetation: 1
realtimeReflectionProbes: 1
billboardsFaceCameraPosition: 1
useLegacyDetailDistribution: 1
vSyncCount: 1
lodBias: 1.5
maximumLODLevel: 0
enableLODCrossFade: 1
streamingMipmapsActive: 0
streamingMipmapsMemoryBudget: 512
streamingMipmapsAddAllCameras: 1
streamingMipmapsRenderersPerFrame: 512
streamingMipmapsMaxLevelReduction: 2
streamingMipmapsMaxFileIORequests: 1024
particles: 1
shadows: 2
shadowsWorkWithLowSettings: 1
vSync: 1
realtimeGICPUUsage: 100
textureQuality: 3
resolutionScalingFixedDPIFactor: 1

```
  customRenderingSettings: 0
  terrainQualityOverrides: 0
  terrainPixelError: 1
  terrainDetailDensityScale: 1
  terrainDetailDensityMultiplier: 1
  terrainBasemapDistance: 1000
  terrainDetailDistance: 200
  terrainTreeDistance: 2000
  terrainBillboardStart: 50
  terrainFadeLength: 5
  terrainMaxTrees: 500
  asyncAssetUploadPersistentBuffer: 1
  asyncAssetUploadBufferSize: 16
  overridePlatformResolution: 0
  resolutionScalingMode: 0
  androidSupportedAspectRatio: 1
  androidMaxAspectRatio: 2.1
  androidBlitType: 0
  androidStartInFullscreen: 1
  androidRenderOutsideSafeArea: 1
  androidUseSwappy: 1
  m_CulturalSettings:
    m_RespectPrayerTimes: 0
    m_PreserveCulturalFidelity: 1
    m_ReducedVisualIntensity: 0
    m_DisableLoudEffects: 0
    m_SimplifiedUI: 0
m_PerPlatformDefaultQuality: {}
```

58. InputActions.asset


```
{
    "name": "RVA_InputActions",
    "maps": [
        {
            "name": "Player",
            "id": "player-controls",
            "actions": [
                {
                    "name": "Move",
                    "type": "Value",
                    "id": "move-action",
                    "expectedControlType": "Vector2",
                    "bindings": [
                        {
                            "name": "WASD",
                            "id": "wasd-binding",
                            "path":
"<Keyboard>/w,<Keyboard>/s,<Keyboard>/a,<Keyboard>/d",
                            "interactions": "",
                            "processors": ""
                        },
                        {
                            "name": "Arrow Keys",
                            "id": "arrows-binding",
```

```
                    "path":
"<Keyboard>/upArrow,<Keyboard>/downArrow,<Keyboard>/leftArrow,<Keyboard
>/rightArrow",
                    "interactions": "",
                    "processors": ""
                },
                {
                    "name": "Gamepad Stick",
                    "id": "gamepad-stick",
                    "path": "<Gamepad>/leftStick",
                    "interactions": "",
                    "processors": "StickDeadzone(min=0.125,max=0.925)"
                },
                {
                    "name": "Touch Joystick",
                    "id": "touch-joystick",
                    "path": "<Touchscreen>/primaryTouch/position",
                    "interactions": "",
                    "processors": "StickDeadzone(min=0.1,max=0.9)"
                }
            ]
        },
        {
            "name": "Look",
            "type": "Value",
            "id": "look-action",
            "expectedControlType": "Vector2",
            "bindings": [
                {
                    "name": "Mouse Delta",
                    "id": "mouse-delta",
```

```
            "path": "<Mouse>/delta",
            "interactions": "",
            "processors": "ScaleVector2(x=0.15,y=0.15)"
        },
        {
            "name": "Gamepad Right Stick",
            "id": "gamepad-look",
            "path": "<Gamepad>/rightStick",
            "interactions": "",
            "processors": "StickDeadzone(min=0.125,max=0.925)"
        },
        {
            "name": "Touch Drag",
            "id": "touch-look",
            "path": "<Touchscreen>/primaryTouch/delta",
            "interactions": "",
            "processors": "ScaleVector2(x=0.01,y=0.01)"
        }
    ]
},
{
    "name": "Run",
    "type": "Button",
    "id": "run-action",
    "expectedControlType": "Button",
    "bindings": [
        {
            "name": "Left Shift",
            "id": "shift-binding",
            "path": "<Keyboard>/leftShift",
            "interactions": "",
```

```
            "processors": ""
        },
        {
            "name": "Gamepad Button",
            "id": "gamepad-run",
            "path": "<Gamepad>/leftShoulder",
            "interactions": "",
            "processors": ""
        },
        {
            "name": "Touch Run",
            "id": "touch-run",
            "path": "<Touchscreen>/secondaryTouch/press",
            "interactions": "",
            "processors": ""
        }
    ]
},
{
    "name": "Jump",
    "type": "Button",
    "id": "jump-action",
    "expectedControlType": "Button",
    "bindings": [
        {
            "name": "Space",
            "id": "space-binding",
            "path": "<Keyboard>/space",
            "interactions": "",
            "processors": ""
        },
```

```
        {
            "name": "Gamepad Button",
            "id": "gamepad-jump",
            "path": "<Gamepad>/buttonSouth",
            "interactions": "",
            "processors": ""
        },
        {
            "name": "Touch Jump",
            "id": "touch-jump",
            "path": "<Touchscreen>/tap",
            "interactions": "Tap",
            "processors": ""
        }
    ]
},
{
    "name": "Interact",
    "type": "Button",
    "id": "interact-action",
    "expectedControlType": "Button",
    "bindings": [
        {
            "name": "E Key",
            "id": "e-binding",
            "path": "<Keyboard>/e",
            "interactions": "",
            "processors": ""
        },
        {
            "name": "Gamepad Button",
```

```
            "id": "gamepad-interact",
            "path": "<Gamepad>/buttonEast",
            "interactions": "",
            "processors": ""
        },
        {
            "name": "Touch Interact",
            "id": "touch-interact",
            "path": "<Touchscreen>/tap",
            "interactions": "Tap",
            "processors": ""
        }
    ]
},
{
    "name": "PrayerMode",
    "type": "Button",
    "id": "prayer-action",
    "expectedControlType": "Button",
    "bindings": [
        {
            "name": "P Key",
            "id": "prayer-key",
            "path": "<Keyboard>/p",
            "interactions": "",
            "processors": ""
        },
        {
            "name": "Gamepad DPad",
            "id": "gamepad-prayer",
            "path": "<Gamepad>/dpad/up",
```

```
                "interactions": "",
                "processors": ""
            }
        ]
    },
    {
        "name": "CulturalUI",
        "type": "Button",
        "id": "cultural-ui",
        "expectedControlType": "Button",
        "bindings": [
            {
                "name": "C Key",
                "id": "cultural-key",
                "path": "<Keyboard>/c",
                "interactions": "",
                "processors": ""
            },
            {
                "name": "Gamepad DPad",
                "id": "gamepad-cultural",
                "path": "<Gamepad>/dpad/down",
                "interactions": "",
                "processors": ""
            }
        ]
    }
],
"controlSchemes": [
    {
        "name": "KeyboardMouse",
```

```
    "id": "keyboard-mouse",
    "deviceRequirements": [
      {
        "controlPath": "<Keyboard>",
        "isOptional": false,
        "isOR": false
      },
      {
        "controlPath": "<Mouse>",
        "isOptional": false,
        "isOR": false
      }
    ]
  },
  {
    "name": "Gamepad",
    "id": "gamepad",
    "deviceRequirements": [
      {
        "controlPath": "<Gamepad>",
        "isOptional": false,
        "isOR": false
      }
    ]
  },
  {
    "name": "Touch",
    "id": "touch",
    "deviceRequirements": [
      {
        "controlPath": "<Touchscreen>",
```

```
                    "isOptional": false,
                    "isOR": false
                }
            ]
        }
    ]
},
{
    "name": "UI",
    "id": "ui-controls",
    "actions": [
        {
            "name": "Navigate",
            "type": "Value",
            "id": "navigate-action",
            "expectedControlType": "Vector2",
            "bindings": [
                {
                    "name": "Arrow Keys",
                    "id": "ui-arrows",
                    "path":
"<Keyboard>/upArrow,<Keyboard>/downArrow,<Keyboard>/leftArrow,<Keyboard
>/rightArrow",
                    "interactions": "",
                    "processors": ""
                },
                {
                    "name": "Gamepad DPad",
                    "id": "ui-dpad",
                    "path": "<Gamepad>/dpad",
                    "interactions": "",
```

```
            "processors": ""
        },
        {
            "name": "Gamepad Left Stick",
            "id": "ui-stick",
            "path": "<Gamepad>/leftStick",
            "interactions": "",
            "processors": ""
        }
    ]
},
{
    "name": "Submit",
    "type": "Button",
    "id": "submit-action",
    "expectedControlType": "Button",
    "bindings": [
        {
            "name": "Enter",
            "id": "enter-binding",
            "path": "<Keyboard>/enter",
            "interactions": "",
            "processors": ""
        },
        {
            "name": "Gamepad South",
            "id": "gamepad-submit",
            "path": "<Gamepad>/buttonSouth",
            "interactions": "",
            "processors": ""
        }
```

```
    ]
},
{
    "name": "Cancel",
    "type": "Button",
    "id": "cancel-action",
    "expectedControlType": "Button",
    "bindings": [
        {
            "name": "Escape",
            "id": "escape-binding",
            "path": "<Keyboard>/escape",
            "interactions": "",
            "processors": ""
        },
        {
            "name": "Gamepad East",
            "id": "gamepad-cancel",
            "path": "<Gamepad>/buttonEast",
            "interactions": "",
            "processors": ""
        }
    ]
},
{
    "name": "Point",
    "type": "PassThrough",
    "id": "point-action",
    "expectedControlType": "Vector2",
    "bindings": [
        {
```

```
            "name": "Mouse Position",
            "id": "mouse-point",
            "path": "<Mouse>/position",
            "interactions": "",
            "processors": ""
        },
        {
            "name": "Touch Position",
            "id": "touch-point",
            "path": "<Touchscreen>/primaryTouch/position",
            "interactions": "",
            "processors": ""
        }
    ]
},
{
    "name": "Click",
    "type": "Button",
    "id": "click-action",
    "expectedControlType": "Button",
    "bindings": [
        {
            "name": "Left Click",
            "id": "mouse-click",
            "path": "<Mouse>/leftButton",
            "interactions": "",
            "processors": ""
        },
        {
            "name": "Touch Tap",
            "id": "touch-tap",
```

```
                "path": "<Touchscreen>/primaryTouch/press",
                "interactions": "",
                "processors": ""
            }
        ]
    },
    {
        "name": "ScrollWheel",
        "type": "PassThrough",
        "id": "scroll-action",
        "expectedControlType": "Vector2",
        "bindings": [
            {
                "name": "Mouse Wheel",
                "id": "mouse-wheel",
                "path": "<Mouse>/scroll",
                "interactions": "",
                "processors": ""
            }
        ]
    }
  ]
},
{
  "name": "Cultural",
  "id": "cultural-controls",
  "actions": [
    {
      "name": "RespectfulMode",
      "type": "Button",
      "id": "respectful-mode",
```

```
    "expectedControlType": "Button",
    "bindings": [
      {
        "name": "R Key",
        "id": "respectful-key",
        "path": "<Keyboard>/r",
        "interactions": "",
        "processors": ""
      }
    ]
  },
  {
    "name": "CommunityInteraction",
    "type": "Button",
    "id": "community-action",
    "expectedControlType": "Button",
    "bindings": [
      {
        "name": "T Key",
        "id": "community-key",
        "path": "<Keyboard>/t",
        "interactions": "",
        "processors": ""
      }
    ]
  },
  {
    "name": "TraditionalGesture",
    "type": "Button",
    "id": "traditional-gesture",
    "expectedControlType": "Button",
```

```
                "bindings": [
                    {
                        "name": "G Key",
                        "id": "gesture-key",
                        "path": "<Keyboard>/g",
                        "interactions": "",
                        "processors": ""
                    }
                ]
            }
        ]
    }
  ],
  "culturalInputSettings": {
    "respectPrayerTimes": true,
    "reduceInputDuringPrayer": true,
    "enableTraditionalGestures": true,
    "disableLoudInputEffects": true,
    "culturalSensitivityMode": "Respectful"
  }
}
```

59. AudioMixer.mixer

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!241 &24100000
```

```
AudioMixerController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: RVA_AudioMixer
  m_OutputGroup: {fileID: 0}
  m_MasterGroup: {fileID: 24300002}
  m_Snapshots:
  - {fileID: 24500006}
  - {fileID: 24500004}
  - {fileID: 24500008}
  - {fileID: 24500010}
  - {fileID: 24500012}
  m_StartSnapshot: {fileID: 24500006}
  m_SuspendThreshold: -80
  m_EnableSuspend: 1
  m_UpdateMode: 0
  m_ExposedParameters:
  - guid: 3f1d65bca1e4c4a4a8f3e8b9c7d5e1f2
    name: MasterVolume
  - guid: a8b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a
    name: MusicVolume
  - guid: b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b
    name: SFXVolume
  - guid: c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b4a
    name: AmbientVolume
  - guid: d5e1f2a4b4a8f3e8b9c7d5e1f2a4b4a8f
    name: PrayerTimeMultiplier
  - guid: e1f2a4b4a8f3e8b9c7d5e1f2a4b4a8f3
    name: CulturalModeVolume
```

```
  m_AudioMixerGroupViews:
  - guids:
    - 24300002
    - 24300004
    - 24300006
    - 24300008
    - 24300010
    - 24300012
    - 24300014
    name: View
  m_CurrentMixerGroupView: View
  m_TargetSnapshot: {fileID: 24500006}
--- !u!243 &24300002
AudioMixerGroupController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: Master
  m_AudioMixer: {fileID: 24100000}
  m_GroupID: 24300002
  m_Children:
  - {fileID: 24300004}
  - {fileID: 24300006}
  - {fileID: 24300008}
  - {fileID: 24300012}
  m_Volume: 3f1d65bca1e4c4a4a8f3e8b9c7d5e1f2
  m_Pitch: 4f2e6d8b9c1a4e4b8d7f5a2c1e6b9f3
  m_Send: 00000000
  m_Effects:
  - {fileID: 24400000}
```

```
  m_UserColorIndex: 0
  m_Mute: 0
  m_Solo: 0
  m_BypassEffects: 0
--- !u!243 &24300004
AudioMixerGroupController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: Music
  m_AudioMixer: {fileID: 24100000}
  m_GroupID: 24300004
  m_Children:
  - {fileID: 24300010}
  - {fileID: 24300014}
  m_Volume: a8b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a
  m_Pitch: 5g3f7e9c2b5f5c5c9e8g6b3d7f0c2a
  m_Send: 00000000
  m_Effects:
  - {fileID: 24400002}
  m_UserColorIndex: 1
  m_Mute: 0
  m_Solo: 0
  m_BypassEffects: 0
--- !u!243 &24300006
AudioMixerGroupController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
```

```
  m_Name: SFX
  m_AudioMixer: {fileID: 24100000}
  m_GroupID: 24300006
  m_Children: []
  m_Volume: b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b
  m_Pitch: 6h4g8f0d3c6g6d6d0f9h7c4e8g1d3b
  m_Send: 00000000
  m_Effects:
  - {fileID: 24400004}
  m_UserColorIndex: 2
  m_Mute: 0
  m_Solo: 0
  m_BypassEffects: 0
--- !u!243 &24300008
AudioMixerGroupController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: Ambient
  m_AudioMixer: {fileID: 24100000}
  m_GroupID: 24300008
  m_Children: []
  m_Volume: c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b4a
  m_Pitch: 7i5h9g1e4d7h7e7e1g0i8d5f9h2e4c
  m_Send: 00000000
  m_Effects:
  - {fileID: 24400006}
  m_UserColorIndex: 3
  m_Mute: 0
  m_Solo: 0
```

```
  m_BypassEffects: 0
--- !u!243 &24300010
AudioMixerGroupController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: TraditionalMusic
  m_AudioMixer: {fileID: 24100000}
  m_GroupID: 24300010
  m_Children: []
  m_Volume: 8j6i0h2f5e8i8f8f2h1j9e6g0i3f5d
  m_Pitch: 9k7j1i3f6g9j9g9g2h1j0f7g1i4e
  m_Send: 00000000
  m_Effects:
  - {fileID: 24400008}
  m_UserColorIndex: 4
  m_Mute: 0
  m_Solo: 0
  m_BypassEffects: 0
--- !u!243 &24300012
AudioMixerGroupController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: UI
  m_AudioMixer: {fileID: 24100000}
  m_GroupID: 24300012
  m_Children: []
  m_Volume: d5e1f2a4b4a8f3e8b9c7d5e1f2a4b4a8f
```

```
  m_Pitch: 0l8k2j4g7h0k0h0h3i2k1g8h2j5f
  m_Send: 00000000
  m_Effects:
  - {fileID: 24400010}
  m_UserColorIndex: 5
  m_Mute: 0
  m_Solo: 0
  m_BypassEffects: 0
--- !u!243 &24300014
AudioMixerGroupController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: ReligiousAudio
  m_AudioMixer: {fileID: 24100000}
  m_GroupID: 24300014
  m_Children: []
  m_Volume: e1f2a4b4a8f3e8b9c7d5e1f2a4b4a8f3
  m_Pitch: 1m9l3k5h8i1l1i1i4j3l2h9i3k6g
  m_Send: 00000000
  m_Effects:
  - {fileID: 24400012}
  m_UserColorIndex: 6
  m_Mute: 0
  m_Solo: 0
  m_BypassEffects: 0
--- !u!244 &24400000
AudioEffectController:
  m_ObjectHideFlags: 3
  m_CorrespondingSourceObject: {fileID: 0}
```

```
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name:
  m_EffectID: 24400000
  m_EffectName: Attenuation
  m_MixLevel: 2f1d65bca1e4c4a4a8f3e8b9c7d5e1f2
  m_Parameters: []
  m_SendTarget: {fileID: 0}
  m_EnableWetMix: 0
  m_Bypass: 0
--- !u!244 &24400002
AudioEffectController:
  m_ObjectHideFlags: 3
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name:
  m_EffectID: 24400002
  m_EffectName: Attenuation
  m_MixLevel: 3g2e76dcb2f5d5b5b9g4f9c8d6e2f3g
  m_Parameters: []
  m_SendTarget: {fileID: 0}
  m_EnableWetMix: 0
  m_Bypass: 0
--- !u!244 &24400004
AudioEffectController:
  m_ObjectHideFlags: 3
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name:
```

```
  m_EffectID: 24400004
  m_EffectName: Attenuation
  m_MixLevel: 4h3f87edc3g6e6c6c0g5g0d9e7f4g4
  m_Parameters: []
  m_SendTarget: {fileID: 0}
  m_EnableWetMix: 0
  m_Bypass: 0
--- !u!244 &24400006
AudioEffectController:
  m_ObjectHideFlags: 3
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name:
  m_EffectID: 24400006
  m_EffectName: Attenuation
  m_MixLevel: 5i4g98fed4h7f7d7d1h6h1e0f8g5h5
  m_Parameters: []
  m_SendTarget: {fileID: 0}
  m_EnableWetMix: 0
  m_Bypass: 0
--- !u!244 &24400008
AudioEffectController:
  m_ObjectHideFlags: 3
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name:
  m_EffectID: 24400008
  m_EffectName: Attenuation
  m_MixLevel: 6j5h09gfe5i8g8e8e2i7i2f1g9h6i6
```

```
  m_Parameters: []
  m_SendTarget: {fileID: 0}
  m_EnableWetMix: 0
  m_Bypass: 0
--- !u!244 &24400010
AudioEffectController:
  m_ObjectHideFlags: 3
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name:
  m_EffectID: 24400010
  m_EffectName: Attenuation
  m_MixLevel: 7k6i1ahgf6j9h9f9f3j8j3g2h0i7j7
  m_Parameters: []
  m_SendTarget: {fileID: 0}
  m_EnableWetMix: 0
  m_Bypass: 0
--- !u!244 &24400012
AudioEffectController:
  m_ObjectHideFlags: 3
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name:
  m_EffectID: 24400012
  m_EffectName: Attenuation
  m_MixLevel: 8l7j2bihg7k0i0g0g4k9k4h3i1j8k8
  m_Parameters: []
  m_SendTarget: {fileID: 0}
  m_EnableWetMix: 0
```

```
  m_Bypass: 0
--- !u!245 &24500004
AudioMixerSnapshotController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: PrayerTime
  m_AudioMixer: {fileID: 24100000}
  m_SnapshotID: 24500004
  m_FloatValues:
    3f1d65bca1e4c4a4a8f3e8b9c7d5e1f2: -15
    a8b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a: -20
    b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b: -25
    c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b4a: -10
    8j6i0h2f5e8i8f8f2h1j9e6g0i3f5d: -5
    e1f2a4b4a8f3e8b9c7d5e1f2a4b4a8f3: -30
  m_TransitionOverrides: {}
--- !u!245 &24500006
AudioMixerSnapshotController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: Normal
  m_AudioMixer: {fileID: 24100000}
  m_SnapshotID: 24500006
  m_FloatValues:
    3f1d65bca1e4c4a4a8f3e8b9c7d5e1f2: 0
    a8b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a: -5
    b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b: 0
```

```
    c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b4a: -5
    8j6i0h2f5e8i8f8f2h1j9e6g0i3f5d: -10
    e1f2a4b4a8f3e8b9c7d5e1f2a4b4a8f3: -15
  m_TransitionOverrides: {}
--- !u!245 &24500008
AudioMixerSnapshotController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: CulturalEvent
  m_AudioMixer: {fileID: 24100000}
  m_SnapshotID: 24500008
  m_FloatValues:
    3f1d65bca1e4c4a4a8f3e8b9c7d5e1f2: -5
    a8b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a: 0
    b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b: -5
    c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b4a: 0
    8j6i0h2f5e8i8f8f2h1j9e6g0i3f5d: -5
    e1f2a4b4a8f3e8b9c7d5e1f2a4b4a8f3: -10
  m_TransitionOverrides: {}
--- !u!245 &24500010
AudioMixerSnapshotController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: CommunityMode
  m_AudioMixer: {fileID: 24100000}
  m_SnapshotID: 24500010
  m_FloatValues:
```

```
    3f1d65bca1e4c4a4a8f3e8b9c7d5e1f2: -3
    a8b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a: -8
    b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b: -3
    c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b4a: -3
    8j6i0h2f5e8i8f8f2h1j9e6g0i3f5d: 0
    e1f2a4b4a8f3e8b9c7d5e1f2a4b4a8f3: -8
  m_TransitionOverrides: {}
--- !u!245 &24500012
AudioMixerSnapshotController:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_Name: RespectfulMode
  m_AudioMixer: {fileID: 24100000}
  m_SnapshotID: 24500012
  m_FloatValues:
    3f1d65bca1e4c4a4a8f3e8b9c7d5e1f2: -8
    a8b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a: -12
    b9c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b: -15
    c7d5e1f2a4b4a8f3e8b9c7d5e1f2a4b4a: -5
    8j6i0h2f5e8i8f8f2h1j9e6g0i3f5d: -8
    e1f2a4b4a8f3e8b9c7d5e1f2a4b4a8f3: -20
  m_TransitionOverrides: {}
```

60. TagManager.asset

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!78 &1
TagManager:
  serializedVersion: 2
  tags:
  - Player
  - Enemy
  - NPC
  - Vehicle
  - Weapon
  - Pickup
  - Interactive
  - UI
  - NonEssentialUI
  - CulturalSite
  - ReligiousSite
  - CommunityArea
  - TraditionalZone
  - PrayerArea
  - Mosque
  - Island
  - Water
  - Beach
  - PalmTree
```

- Building
- Road
- VehiclePath
- SpawnPoint
- SavePoint
- Checkpoint
- MissionObjective
- Collectible
- AudioSource
- WeatherSystem
- PrayerTimeNotifier
- CulturalEvent
- TraditionalMusic
- ReligiousAudio
- BoduberuDrum
- FishingSpot
- MarketStall
- CommunityGathering
- MaldivianFlag
- IslamicSymbol
- TraditionalDress
- LocalCraft
- CoconutTree
- CoralReef
- Lagoon
- Harbor
- Airport
- Resort
- GovernmentBuilding
- School
- Hospital

- Shop
- Restaurant
- Cafe
- GuestHouse
- LocalHouse
- HistoricalSite
- AncientRuins
- TraditionalWell
- CommunityCenter
- SportsField
- Playground
- Park
- Garden
- FarmLand
- FishingBoat
- SpeedBoat
- Yacht
- Seaplane
- JetSki
- TraditionalBoat
- Dhoni
- Banca
- FishingNet
- Anchor
- Lighthouse
- Buoy
- Wave
- Current
- Tide
- WindZone
- RainZone

- Cloud
- Sun
- Moon
- Star
- PrayerCall
- Azan
- MosqueBell
- TraditionalAnnouncement
- CommunityDrum
- CelebrationMusic
- WeddingMusic
- FuneralBell
- RespectfulSilence
- CulturalWarning
- ReligiousWarning
- CommunityNotification
- TraditionalTeaching
- IslamicTeaching
- LocalWisdom
- EnvironmentalWarning
- WeatherAlert
- OceanWarning
- StormWarning
- HighTideWarning
- LowTideAlert
- FishingAlert
- CommunityMeeting
- ReligiousGathering
- CulturalFestival
- TraditionalCeremony
- IslamicHoliday

- NationalHoliday
- LocalFestival
- TraditionalDance
- BoduberuPerformance
- LocalTheater
- StoryTelling
- TraditionalGame
- CommunitySport
- FishingCompetition
- CookingCompetition
- CraftCompetition
- TraditionalRace
- BoatRace
- SwimmingCompetition
- VolleyballMatch
- FootballMatch
- CricketMatch
- TraditionalSport
- RespectfulArea
- QuietZone
- NoMusicZone
- ReducedVolumeZone
- CulturalSensitivityZone
- ReligiousRespectZone
- CommunityHarmonyZone
- TraditionalPreservationZone
- EnvironmentalProtectionZone
- MarineConservationZone
- CulturalHeritageZone
- IslamicHeritageZone
- LocalTraditionZone

- CommunityValueZone
- RespectfulInteractionZone
- PrayerRespectZone
- CulturalLearningZone
- TraditionalTeachingZone
- IslamicTeachingZone
- EnvironmentalEducationZone
- CommunityEducationZone
- LocalWisdomZone
- CulturalExchangeZone
- TraditionalCraftZone
- IslamicArtZone
- LocalArtZone
- CommunityArtZone
- TraditionalMusicZone
- IslamicMusicZone
- LocalMusicZone
- CommunityMusicZone
- RespectfulSoundZone
- QuietSoundZone
- TraditionalSoundZone
- IslamicSoundZone
- LocalSoundZone
- CommunitySoundZone
- EnvironmentalSoundZone
- MarineSoundZone
- NaturalSoundZone
- CulturalSoundZone
- ReligiousSoundZone
- TraditionalSoundZone
- CommunitySoundZone

- RespectfulAudioZone
- QuietAudioZone
- CulturalAudioZone
- IslamicAudioZone
- LocalAudioZone
- CommunityAudioZone
- EnvironmentalAudioZone
- MarineAudioZone
- NaturalAudioZone
- TraditionalAudioZone
- ReligiousAudioZone
- RespectfulAudioZone

layers:
- Default
- TransparentFX
- Ignore Raycast
- Water
- UI
- Player
- Enemy
- NPC
- Vehicle
- Weapon
- Pickup
- Interactive
- CulturalSite
- ReligiousSite
- CommunityArea
- TraditionalZone
- PrayerArea
- Mosque

- Island
- Water
- Beach
- PalmTree
- Building
- Road
- VehiclePath
- SpawnPoint
- SavePoint
- Checkpoint
- MissionObjective
- Collectible
- AudioSource
- WeatherSystem
- PrayerTimeNotifier
- CulturalEvent
- TraditionalMusic
- ReligiousAudio
- BoduberuDrum
- FishingSpot
- MarketStall
- CommunityGathering
- MaldivianFlag
- IslamicSymbol
- TraditionalDress
- LocalCraft
- CoconutTree
- CoralReef
- Lagoon
- Harbor
- Airport

- Resort
- GovernmentBuilding
- School
- Hospital
- Shop
- Restaurant
- Cafe
- GuestHouse
- LocalHouse
- HistoricalSite
- AncientRuins
- TraditionalWell
- CommunityCenter
- SportsField
- Playground
- Park
- Garden
- FarmLand
- FishingBoat
- SpeedBoat
- Yacht
- Seaplane
- JetSki
- TraditionalBoat
- Dhoni
- Banca
- FishingNet
- Anchor
- Lighthouse
- Buoy
- Wave

- Current
- Tide
- WindZone
- RainZone
- Cloud
- Sun
- Moon
- Star
- PrayerCall
- Azan
- MosqueBell
- TraditionalAnnouncement
- CommunityDrum
- CelebrationMusic
- WeddingMusic
- FuneralBell
- RespectfulSilence
- CulturalWarning
- ReligiousWarning
- CommunityNotification
- TraditionalTeaching
- IslamicTeaching
- LocalWisdom
- EnvironmentalWarning
- WeatherAlert
- OceanWarning
- StormWarning
- HighTideWarning
- LowTideAlert
- FishingAlert
- CommunityMeeting

- ReligiousGathering
- CulturalFestival
- TraditionalCeremony
- IslamicHoliday
- NationalHoliday
- LocalFestival
- TraditionalDance
- BoduberuPerformance
- LocalTheater
- StoryTelling
- TraditionalGame
- CommunitySport
- FishingCompetition
- CookingCompetition
- CraftCompetition
- TraditionalRace
- BoatRace
- SwimmingCompetition
- VolleyballMatch
- FootballMatch
- CricketMatch
- TraditionalSport
- RespectfulArea
- QuietZone
- NoMusicZone
- ReducedVolumeZone
- CulturalSensitivityZone
- ReligiousRespectZone
- CommunityHarmonyZone
- TraditionalPreservationZone
- EnvironmentalProtectionZone

- MarineConservationZone
- CulturalHeritageZone
- IslamicHeritageZone
- LocalTraditionZone
- CommunityValueZone
- RespectfulInteractionZone
- PrayerRespectZone
- CulturalLearningZone
- TraditionalTeachingZone
- IslamicTeachingZone
- EnvironmentalEducationZone
- CommunityEducationZone
- LocalWisdomZone
- CulturalExchangeZone
- TraditionalCraftZone
- IslamicArtZone
- LocalArtZone
- CommunityArtZone
- TraditionalMusicZone
- IslamicMusicZone
- LocalMusicZone
- CommunityMusicZone
- RespectfulSoundZone
- QuietSoundZone
- TraditionalSoundZone
- IslamicSoundZone
- LocalSoundZone
- CommunitySoundZone
- EnvironmentalSoundZone
- MarineSoundZone
- NaturalSoundZone

- CulturalSoundZone
- ReligiousSoundZone
- TraditionalSoundZone
- CommunitySoundZone
- RespectfulAudioZone
- QuietAudioZone
- CulturalAudioZone
- IslamicAudioZone
- LocalAudioZone
- CommunityAudioZone
- EnvironmentalAudioZone
- MarineAudioZone
- NaturalAudioZone
- TraditionalAudioZone
- ReligiousAudioZone
- RespectfulAudioZone

m_SortingLayers:
- name: Default
  uniqueID: 0
  locked: 0
- name: Background
  uniqueID: 2984371629
  locked: 0
- name: CulturalBackground
  uniqueID: 2984371630
  locked: 0
- name: Environment
  uniqueID: 2984371631
  locked: 0
- name: CulturalEnvironment
  uniqueID: 2984371632

```
  locked: 0
- name: Props
  uniqueID: 2984371633
  locked: 0
- name: CulturalProps
  uniqueID: 2984371634
  locked: 0
- name: Characters
  uniqueID: 2984371635
  locked: 0
- name: CulturalCharacters
  uniqueID: 2984371636
  locked: 0
- name: Vehicles
  uniqueID: 2984371637
  locked: 0
- name: TraditionalVehicles
  uniqueID: 2984371638
  locked: 0
- name: Effects
  uniqueID: 2984371639
  locked: 0
- name: CulturalEffects
  uniqueID: 2984371640
  locked: 0
- name: UI
  uniqueID: 2984371641
  locked: 0
- name: CulturalUI
  uniqueID: 2984371642
  locked: 0
```

- name: PrayerUI
  uniqueID: 2984371643
  locked: 0
- name: CommunityUI
  uniqueID: 2984371644
  locked: 0
- name: TraditionalUI
  uniqueID: 2984371645
  locked: 0
- name: ReligiousUI
  uniqueID: 2984371646
  locked: 0
- name: Foreground
  uniqueID: 2984371647
  locked: 0
- name: CulturalForeground
  uniqueID: 2984371648
  locked: 0
- name: Overlay
  uniqueID: 2984371649
  locked: 0
- name: CulturalOverlay
  uniqueID: 2984371650
  locked: 0
- name: PrayerOverlay
  uniqueID: 2984371651
  locked: 0
- name: CommunityOverlay
  uniqueID: 2984371652
  locked: 0
- name: TraditionalOverlay

```
    uniqueID: 2984371653
    locked: 0
  - name: ReligiousOverlay
    uniqueID: 2984371654
    locked: 0
```

61. Physics2DSettings.asset

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!19 &1
Physics2DSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 5
  m_Gravity: {x: 0, y: -9.81}
  m_DefaultMaterial: {fileID: 0}
  m_VelocityIterations: 8
  m_PositionIterations: 3
  m_VelocityThreshold: 1
  m_MaxLinearCorrection: 0.2
  m_MaxAngularCorrection: 8
  m_MaxTranslationSpeed: 100
  m_MaxRotationSpeed: 360
  m_MinPenetrationForPenalty: 0.01
  m_BaumgarteScale: 0.2
  m_BaumgarteTimeOfImpactScale: 0.75
  m_TimeToSleep: 0.5
```

m_LinearSleepingTolerance: 0.01
m_AngularSleepingTolerance: 2
m_DefaultContactOffset: 0.01
m_JobOptions:
  serializedVersion: 2
  useMultithreading: 1
  useConsistencySorting: 1
  m_InterpolationPosesPerJob: 100
  m_NewContactsPerJob: 30
  m_CollideContactsPerJob: 100
  m_ClearFlagsPerJob: 200
  m_ClearBodyForcesPerJob: 500
  m_SyncDiscreteFixturesPerJob: 50
  m_SyncContinuousFixturesPerJob: 50
  m_FindNearestContactsPerJob: 100
  m_UpdateTriggerContactsPerJob: 100
  m_IslandSolverCostThreshold: 100
  m_IslandSolverBodyCostScale: 1
  m_IslandSolverContactCostScale: 10
  m_IslandSolverJointCostScale: 10
  m_IslandSolverBodiesPerJob: 50
  m_IslandSolverContactsPerJob: 50
m_SimulationMode: 0
m_SimulationLayers:
  serializedVersion: 2
  m_Bits: 4294967295
m_MaxSubSteps: 4
m_MinSubSteps: 1
m_ReuseCollisionCallbacks: 1
m_QueriesHitTriggers: 1
m_QueriesStartInColliders: 1

```
  m_CallbacksOnDisable: 1
  m_ReuseCollisionCallbacksOnDisable: 1
  m_AutoSyncTransforms: 0
  m_AlwaysShowCollisions: 0
  m_ShowColliderSleep: 0
  m_ShowColliderContacts: 0
  m_ShowColliderAABB: 0
  m_ContactArrowScale: 0.2
  m_ColliderAwakeColor: {r: 0.5686275, g: 0.95686275, b: 0.54509807, a:
0.7058824}
  m_ColliderAsleepColor: {r: 0.5686275, g: 0.95686275, b: 0.54509807, a:
0.3137255}
  m_ColliderContactColor: {r: 1, g: 0, b: 1, a: 0.6862745}
  m_ColliderAABBColor: {r: 1, g: 1, b: 0, a: 0.2509804}
  m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  m_CulturalPhysicsSettings:
    m_RespectPrayerTimes: 1
    m_ReducedPhysicsDuringPrayer: 1
    m_DisableLoudCollisionSounds: 1
    m_SimplifiedPhysicsDuringCulturalEvents: 1
    m_PreserveTraditionalObjectPhysics: 1
    m_CommunityFriendlyPhysics: 1
    m_ReligiousSitePhysicsRespect: 1
    m_EnvironmentalPhysicsAwareness: 1
    m_MarineLifePhysicsProtection: 1
    m_CulturalHeritagePhysicsPreservation: 1
  m_LayerCollisionPresets:
    Default:
```

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CulturalSite:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  ReligiousSite:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CommunityArea:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  TraditionalZone:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  PrayerArea:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Mosque:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Island:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Water:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Beach:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

PalmTree:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Building:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Road:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

VehiclePath:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

SpawnPoint:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

SavePoint:

m_LayerCollisionMatrix: ff
Checkpoint:
m_LayerCollisionMatrix: ff
MissionObjective:
m_LayerCollisionMatrix: ff
Collectible:
m_LayerCollisionMatrix: ff
AudioSource:
m_LayerCollisionMatrix: ff
WeatherSystem:
m_LayerCollisionMatrix: ff
PrayerTimeNotifier:
m_LayerCollisionMatrix: ff
CulturalEvent:
m_LayerCollisionMatrix: ff

TraditionalMusic:
  m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ReligiousAudio:
  m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
BoduberuDrum:
  m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
FishingSpot:
  m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
MarketStall:
  m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CommunityGathering:
  m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
MaldivianFlag:
  m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
IslamicSymbol:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  TraditionalDress:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  LocalCraft:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CoconutTree:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CoralReef:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Lagoon:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Harbor:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Airport:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Resort:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

GovernmentBuilding:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

School:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Hospital:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Shop:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Restaurant:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Cafe:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

GuestHouse:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  LocalHouse:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  HistoricalSite:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  AncientRuins:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  TraditionalWell:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CommunityCenter:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  SportsField:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Playground:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Park:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Garden:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

FarmLand:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

FishingBoat:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

SpeedBoat:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Yacht:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Seaplane:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

JetSki:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  TraditionalBoat:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Dhoni:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Banca:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  FishingNet:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Anchor:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Lighthouse:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Buoy:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Wave:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Current:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Tide:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

WindZone:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

RainZone:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Cloud:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Sun:

m_LayerCollisionMatrix:

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Moon:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Star:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  PrayerCall:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  Azan:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  MosqueBell:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  TraditionalAnnouncement:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CommunityDrum:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CelebrationMusic:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

WeddingMusic:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
FuneralBell:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
RespectfulSilence:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CulturalWarning:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ReligiousWarning:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CommunityNotification:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
TraditionalTeaching:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
IslamicTeaching:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

LocalWisdom:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

EnvironmentalWarning:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

WeatherAlert:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

OceanWarning:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

StormWarning:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

HighTideWarning:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

LowTideAlert:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

FishingAlert:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CommunityMeeting:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ReligiousGathering:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CulturalFestival:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
TraditionalCeremony:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
IslamicHoliday:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
NationalHoliday:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
LocalFestival:

```
    m_LayerCollisionMatrix: fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  TraditionalDance:
    m_LayerCollisionMatrix: fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  BoduberuPerformance:
    m_LayerCollisionMatrix: fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  LocalTheater:
    m_LayerCollisionMatrix: fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  StoryTelling:
    m_LayerCollisionMatrix: fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  TraditionalGame:
    m_LayerCollisionMatrix: fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CommunitySport:
    m_LayerCollisionMatrix: fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  FishingCompetition:
    m_LayerCollisionMatrix: fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

CookingCompetition:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CraftCompetition:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
TraditionalRace:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
BoatRace:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
SwimmingCompetition:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
VolleyballMatch:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
FootballMatch:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CricketMatch:

    m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  TraditionalSport:
    m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  RespectfulArea:
    m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  QuietZone:
    m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  NoMusicZone:
    m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  ReducedVolumeZone:
    m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CulturalSensitivityZone:
    m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  ReligiousRespectZone:
    m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

CommunityHarmonyZone:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

TraditionalPreservationZone:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

EnvironmentalProtectionZone:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

MarineConservationZone:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

CulturalHeritageZone:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

IslamicHeritageZone:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

LocalTraditionZone:

m_LayerCollisionMatrix:

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

CommunityValueZone:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

  RespectfulInteractionZone:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

  PrayerRespectZone:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

  CulturalLearningZone:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

  TraditionalTeachingZone:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

  IslamicTeachingZone:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

  EnvironmentalEducationZone:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

  CommunityEducationZone:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

LocalWisdomZone:
  m_LayerCollisionMatrix: ff

CulturalExchangeZone:
  m_LayerCollisionMatrix: ff

TraditionalCraftZone:
  m_LayerCollisionMatrix: ff

IslamicArtZone:
  m_LayerCollisionMatrix: ff

LocalArtZone:
  m_LayerCollisionMatrix: ff

CommunityArtZone:
  m_LayerCollisionMatrix: ff

TraditionalMusicZone:
  m_LayerCollisionMatrix: ff

IslamicMusicZone:

m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
LocalMusicZone:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CommunityMusicZone:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
RespectfulSoundZone:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
QuietSoundZone:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
TraditionalSoundZone:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
IslamicSoundZone:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
LocalSoundZone:
m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

CommunitySoundZone:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
EnvironmentalSoundZone:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
MarineSoundZone:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
NaturalSoundZone:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CulturalSoundZone:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ReligiousSoundZone:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
TraditionalSoundZone:
  m_LayerCollisionMatrix:
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
CommunitySoundZone:

    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  RespectfulAudioZone:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  QuietAudioZone:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CulturalAudioZone:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  IslamicAudioZone:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  LocalAudioZone:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  CommunityAudioZone:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  EnvironmentalAudioZone:
    m_LayerCollisionMatrix:
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

    MarineAudioZone:
      m_LayerCollisionMatrix: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    NaturalAudioZone:
      m_LayerCollisionMatrix: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    TraditionalAudioZone:
      m_LayerCollisionMatrix: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    ReligiousAudioZone:
      m_LayerCollisionMatrix: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    RespectfulAudioZone:
      m_LayerCollisionMatrix: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  m_SleepThreshold: 0.01
  m_DefaultLinearDamping: 0.01
  m_DefaultAngularDamping: 0.05
  m_DefaultGravityScale: 1
  m_DefaultMaterial: {fileID: 0}
  m_VelocityThreshold: 1
  m_MaxLinearCorrection: 0.2
  m_MaxAngularCorrection: 8
  m_MaxTranslationSpeed: 100
  m_MaxRotationSpeed: 360
  m_BaumgarteScale: 0.2

```
  m_BaumgarteTimeOfImpactScale: 0.75
  m_DefaultContactOffset: 0.01
  m_JobOptions:
    serializedVersion: 2
    useMultithreading: 1
    useConsistencySorting: 1
    m_InterpolationPosesPerJob: 100
    m_NewContactsPerJob: 30
    m_CollideContactsPerJob: 100
    m_ClearFlagsPerJob: 200
    m_ClearBodyForcesPerJob: 500
    m_SyncDiscreteFixturesPerJob: 50
    m_SyncContinuousFixturesPerJob: 50
    m_FindNearestContactsPerJob: 100
    m_UpdateTriggerContactsPerJob: 100
    m_IslandSolverCostThreshold: 100
    m_IslandSolverBodyCostScale: 1
    m_IslandSolverContactCostScale: 10
    m_IslandSolverJointCostScale: 10
    m_IslandSolverBodiesPerJob: 50
    m_IslandSolverContactsPerJob: 50
```

62. MainMenu.unity

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
```

```
--- !u!29 &1
OcclusionCullingSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 2
  m_OcclusionBakeSettings:
    smallestOccluder: 5
    smallestHole: 0.25
    backfaceThreshold: 100
  m_SceneGUID: 00000000000000000000000000000000
  m_OcclusionCullingData: {fileID: 0}
--- !u!104 &2
RenderSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 9
  m_Fog: 0
  m_FogColor: {r: 0.5, g: 0.5, b: 0.5, a: 1}
  m_FogMode: 3
  m_FogDensity: 0.01
  m_LinearFogStart: 0
  m_LinearFogEnd: 300
  m_AmbientSkyColor: {r: 0.212, g: 0.227, b: 0.259, a: 1}
  m_AmbientEquatorColor: {r: 0.114, g: 0.125, b: 0.133, a: 1}
  m_AmbientGroundColor: {r: 0.047, g: 0.043, b: 0.035, a: 1}
  m_AmbientIntensity: 1
  m_AmbientMode: 0
  m_SubtractiveShadowColor: {r: 0.42, g: 0.478, b: 0.627, a: 1}
  m_SkyboxMaterial: {fileID: 10304, guid:
0000000000000000f000000000000000, type: 0}
  m_HaloStrength: 0.5
  m_FlareStrength: 1
  m_FlareFadeSpeed: 3
```

```
  m_HaloTexture: {fileID: 0}
  m_SpotCookie: {fileID: 10001, guid: 0000000000000000e000000000000000,
type: 0}
  m_DefaultReflectionMode: 0
  m_DefaultReflectionResolution: 128
  m_ReflectionBounces: 1
  m_ReflectionIntensity: 1
  m_CustomReflection: {fileID: 0}
  m_Sun: {fileID: 0}
  m_IndirectSpecularColor: {r: 0.44657898, g: 0.4964133, b: 0.5748178, a: 1}
  m_UseRadianceAmbientProbe: 0
--- !u!157 &3
LightmapSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 12
  m_GIWorkflowMode: 1
  m_GISettings:
    serializedVersion: 2
    m_BounceScale: 1
    m_IndirectOutputScale: 1
    m_AlbedoBoost: 1
    m_EnvironmentLightingMode: 0
    m_EnableBakedLightmaps: 1
    m_EnableRealtimeLightmaps: 0
  m_LightmapEditorSettings:
    serializedVersion: 12
    m_Resolution: 2
    m_BakeResolution: 40
    m_AtlasSize: 1024
    m_AO: 0
    m_AOMaxDistance: 1
```

m_CompAOExponent: 1
m_CompAOExponentDirect: 0
m_ExtractAmbientOcclusion: 0
m_Padding: 2
m_LightmapParameters: {fileID: 0}
m_LightmapsBakeMode: 1
m_TextureCompression: 1
m_FinalGather: 0
m_FinalGatherFiltering: 1
m_FinalGatherRayCount: 256
m_ReflectionCompression: 2
m_MixedBakeMode: 2
m_BakeBackend: 1
m_PVRSampling: 1
m_PVRDirectSampleCount: 32
m_PVRSampleCount: 512
m_PVRBounces: 2
m_PVREnvironmentSampleCount: 256
m_PVREnvironmentReferencePointCount: 2048
m_PVRFilteringMode: 1
m_PVRDenoiserTypeDirect: 1
m_PVRDenoiserTypeIndirect: 1
m_PVRDenoiserTypeAO: 1
m_PVRFilterTypeDirect: 0
m_PVRFilterTypeIndirect: 0
m_PVRFilterTypeAO: 0
m_PVREnvironmentMIS: 1
m_PVRCulling: 1
m_PVRFilteringGaussRadiusDirect: 1
m_PVRFilteringGaussRadiusIndirect: 5
m_PVRFilteringGaussRadiusAO: 2

```
    m_PVRFilteringAtrousPositionSigmaDirect: 0.5
    m_PVRFilteringAtrousPositionSigmaIndirect: 2
    m_PVRFilteringAtrousPositionSigmaAO: 1
    m_ExportTrainingData: 0
    m_TrainingDataDestination: TrainingData
    m_LightProbeSampleCountMultiplier: 4
  m_LightingDataAsset: {fileID: 0}
  m_LightingSettings: {fileID: 0}
--- !u!196 &4
NavMeshSettings:
  serializedVersion: 2
  m_ObjectHideFlags: 0
  m_BuildSettings:
    serializedVersion: 2
    agentTypeID: 0
    agentRadius: 0.5
    agentHeight: 2
    agentSlope: 45
    agentClimb: 0.4
    ledgeDropHeight: 0
    maxJumpAcrossDistance: 0
    minRegionArea: 2
    manualCellSize: 0
    cellSize: 0.16666667
    manualTileSize: 0
    tileSize: 256
    accuratePlacement: 0
    maxJobWorkers: 0
    preserveTilesOutsideBounds: 0
    debug:
      m_Flags: 0
```

```
  m_NavMeshData: {fileID: 0}
--- !u!1 &7055079
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 7055081}
  - component: {fileID: 7055080}
  m_Layer: 0
  m_Name: Directional Light
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!108 &7055080
Light:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 7055079}
  m_Enabled: 1
  serializedVersion: 10
  m_Type: 1
  m_Shape: 0
  m_Color: {r: 1, g: 0.95686275, b: 0.8392157, a: 1}
  m_Intensity: 1
```

```
m_Range: 10
m_SpotAngle: 30
m_InnerSpotAngle: 21.80208
m_CookieSize: 10
m_Shadows:
  m_Type: 2
  m_Resolution: -1
  m_CustomResolution: -1
  m_Strength: 1
  m_Bias: 0.05
  m_NormalBias: 0.4
  m_NearPlane: 0.2
  m_CullingMatrixOverride:
    e00: 1
    e01: 0
    e02: 0
    e03: 0
    e10: 0
    e11: 1
    e12: 0
    e13: 0
    e20: 0
    e21: 0
    e22: 1
    e23: 0
    e30: 0
    e31: 0
    e32: 0
    e33: 1
  m_UseCullingMatrixOverride: 0
m_Cookie: {fileID: 0}
```

```
  m_DrawHalo: 0
  m_Flare: {fileID: 0}
  m_RenderMode: 0
  m_CullingMask:
    serializedVersion: 2
    m_Bits: 4294967295
  m_RenderingLayerMask: 1
  m_Lightmapping: 1
  m_LightShadowCasterMode: 0
  m_AreaSize: {x: 1, y: 1}
  m_BounceIntensity: 1
  m_ColorTemperature: 6570
  m_UseColorTemperature: 0
  m_BoundingSphereOverride: {x: 0, y: 0, z: 0, w: 0}
  m_UseBoundingSphereOverride: 0
  m_UseViewFrustumForShadowCasterCull: 1
  m_ShadowRadius: 0
  m_ShadowAngle: 0
--- !u!4 &7055081
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 7055079}
  m_LocalRotation: {x: 0.40821788, y: -0.23456968, z: 0.10938163, w:
0.8754261}
  m_LocalPosition: {x: 0, y: 3, z: 0}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
```

```
  m_Father: {fileID: 0}
  m_RootOrder: 1
  m_LocalEulerAnglesHint: {x: 50, y: -30, z: 0}
--- !u!1 &963194225
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 963194228}
  - component: {fileID: 963194227}
  - component: {fileID: 963194226}
  m_Layer: 0
  m_Name: Main Camera
  m_TagString: MainCamera
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!81 &963194226
AudioListener:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_Enabled: 1
--- !u!20 &963194227
Camera:
```

m_ObjectHideFlags: 0
m_CorrespondingSourceObject: {fileID: 0}
m_PrefabInstance: {fileID: 0}
m_PrefabAsset: {fileID: 0}
m_GameObject: {fileID: 963194225}
m_Enabled: 1
serializedVersion: 2
m_ClearFlags: 1
m_BackGroundColor: {r: 0.19215687, g: 0.3019608, b: 0.4745098, a: 0}
m_projectionMatrixMode: 1
m_GateFitMode: 2
m_FOVAxisMode: 0
m_SensorSize: {x: 36, y: 24}
m_LensShift: {x: 0, y: 0}
m_FocalLength: 50
m_NormalizedViewPortRect:
  serializedVersion: 2
  x: 0
  y: 0
  width: 1
  height: 1
m_near: 0.3
m_far: 1000
m_fieldOfView: 60
m_orthographic: 0
m_orthographicSize: 5
m_Depth: -1
m_CullingMask:
  serializedVersion: 2
  m_Bits: 4294967295
m_RenderingPath: -1

```
  m_TargetTexture: {fileID: 0}
  m_TargetDisplay: 0
  m_TargetEye: 3
  m_HDR: 1
  m_AllowMSAA: 1
  m_AllowDynamicResolution: 0
  m_ForceIntoRT: 0
  m_OcclusionCulling: 1
  m_StereoConvergence: 10
  m_StereoSeparation: 0.022
--- !u!4 &963194228
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: 0, y: 1, z: -10}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 0
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
--- !u!1 &1234567890
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
```

```
  serializedVersion: 6
  m_Component:
  - component: {fileID: 1234567891}
  - component: {fileID: 1234567892}
  m_Layer: 5
  m_Name: MainMenuUI
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!224 &1234567891
RectTransform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: 0, y: 0, z: 0}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 2
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
  m_AnchorMin: {x: 0.5, y: 0.5}
  m_AnchorMax: {x: 0.5, y: 0.5}
  m_AnchoredPosition: {x: 0, y: 0}
  m_SizeDelta: {x: 100, y: 100}
  m_Pivot: {x: 0.5, y: 0.5}
```

```
--- !u!114 &1234567892
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 7a3c5c661c3e4b4a8f3e8b9c7d5e1f2a, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  gameTitle: "RAAJJE VAGU AUTO: THE ALBAKO CHRONICLES"
  playButtonText: "Start Journey"
  optionsButtonText: "Settings"
  exitButtonText: "Exit Game"
  culturalModeButtonText: "Cultural Mode"
  prayerTimeDisplay: {fileID: 0}
  islamicDateDisplay: {fileID: 0}
  maldivianFlagDisplay: {fileID: 0}
  backgroundVideoPlayer: {fileID: 0}
  traditionalMusicPlayer: {fileID: 0}
  ambientOceanSounds: {fileID: 0}
  mainMenuCanvas: {fileID: 0}
  culturalSettingsPanel: {fileID: 0}
  languageSelectionDropdown: {fileID: 0}
  respectPrayerTimesToggle: {fileID: 0}
  culturalSensitivityModeDropdown: {fileID: 0}
  audioSettingsPanel: {fileID: 0}
  graphicsSettingsPanel: {fileID: 0}
```

```
  gameplaySettingsPanel: {fileID: 0}
  culturalSettingsPanel: {fileID: 0}
  versionText: "Version 1.0.0 - Maldivian Cultural Edition"
```

63. GameWorld.unity

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!29 &1
OcclusionCullingSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 2
  m_OcclusionBakeSettings:
    smallestOccluder: 5
    smallestHole: 0.25
    backfaceThreshold: 100
  m_SceneGUID: 00000000000000000000000000000000
  m_OcclusionCullingData: {fileID: 0}
--- !u!104 &2
RenderSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 9
  m_Fog: 1
  m_FogColor: {r: 0.65882355, g: 0.80784315, b: 0.8745098, a: 1}
  m_FogMode: 3
  m_FogDensity: 0.002
  m_LinearFogStart: 0
  m_LinearFogEnd: 300
```

```
  m_AmbientSkyColor: {r: 0.48, g: 0.69, b: 0.85, a: 1}
  m_AmbientEquatorColor: {r: 0.39, g: 0.49, b: 0.59, a: 1}
  m_AmbientGroundColor: {r: 0.12, g: 0.15, b: 0.18, a: 1}
  m_AmbientIntensity: 1.2
  m_AmbientMode: 0
  m_SubtractiveShadowColor: {r: 0.42, g: 0.478, b: 0.627, a: 1}
  m_SkyboxMaterial: {fileID: 10304, guid:
000000000000000f0000000000000000, type: 0}
  m_HaloStrength: 0.5
  m_FlareStrength: 1
  m_FlareFadeSpeed: 3
  m_HaloTexture: {fileID: 0}
  m_SpotCookie: {fileID: 10001, guid: 0000000000000000e000000000000000,
type: 0}
  m_DefaultReflectionMode: 0
  m_DefaultReflectionResolution: 128
  m_ReflectionBounces: 1
  m_ReflectionIntensity: 1
  m_CustomReflection: {fileID: 0}
  m_Sun: {fileID: 7055079}
  m_IndirectSpecularColor: {r: 0.44657898, g: 0.4964133, b: 0.5748178, a: 1}
  m_UseRadianceAmbientProbe: 0
--- !u!157 &3
LightmapSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 12
  m_GIWorkflowMode: 1
  m_GISettings:
    serializedVersion: 2
    m_BounceScale: 1
    m_IndirectOutputScale: 1
```

```
  m_AlbedoBoost: 1
  m_EnvironmentLightingMode: 0
  m_EnableBakedLightmaps: 1
  m_EnableRealtimeLightmaps: 0
m_LightmapEditorSettings:
  serializedVersion: 12
  m_Resolution: 2
  m_BakeResolution: 40
  m_AtlasSize: 1024
  m_AO: 0
  m_AOMaxDistance: 1
  m_CompAOExponent: 1
  m_CompAOExponentDirect: 0
  m_ExtractAmbientOcclusion: 0
  m_Padding: 2
  m_LightmapParameters: {fileID: 0}
  m_LightmapsBakeMode: 1
  m_TextureCompression: 1
  m_FinalGather: 0
  m_FinalGatherFiltering: 1
  m_FinalGatherRayCount: 256
  m_ReflectionCompression: 2
  m_MixedBakeMode: 2
  m_BakeBackend: 1
  m_PVRSampling: 1
  m_PVRDirectSampleCount: 32
  m_PVRSampleCount: 512
  m_PVRBounces: 2
  m_PVREnvironmentSampleCount: 256
  m_PVREnvironmentReferencePointCount: 2048
  m_PVRFilteringMode: 1
```

```
    m_PVRDenoiserTypeDirect: 1
    m_PVRDenoiserTypeIndirect: 1
    m_PVRDenoiserTypeAO: 1
    m_PVRFilterTypeDirect: 0
    m_PVRFilterTypeIndirect: 0
    m_PVRFilterTypeAO: 0
    m_PVREnvironmentMIS: 1
    m_PVRCulling: 1
    m_PVRFilteringGaussRadiusDirect: 1
    m_PVRFilteringGaussRadiusIndirect: 5
    m_PVRFilteringAtrousPositionSigmaDirect: 0.5
    m_PVRFilteringAtrousPositionSigmaIndirect: 2
    m_PVRFilteringAtrousPositionSigmaAO: 1
    m_ExportTrainingData: 0
    m_TrainingDataDestination: TrainingData
    m_LightProbeSampleCountMultiplier: 4
  m_LightingDataAsset: {fileID: 0}
  m_LightingSettings: {fileID: 0}
--- !u!196 &4
NavMeshSettings:
  serializedVersion: 2
  m_ObjectHideFlags: 0
  m_BuildSettings:
    serializedVersion: 2
    agentTypeID: 0
    agentRadius: 0.5
    agentHeight: 2
    agentSlope: 45
    agentClimb: 0.4
    ledgeDropHeight: 0
    maxJumpAcrossDistance: 0
```

```
    minRegionArea: 2
    manualCellSize: 0
    cellSize: 0.16666667
    manualTileSize: 0
    tileSize: 256
    accuratePlacement: 0
    maxJobWorkers: 0
    preserveTilesOutsideBounds: 0
    debug:
      m_Flags: 0
  m_NavMeshData: {fileID: 0}
--- !u!1 &7055079
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 7055081}
  - component: {fileID: 7055080}
  m_Layer: 0
  m_Name: Sun
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!108 &7055080
Light:
  m_ObjectHideFlags: 0
```

m_CorrespondingSourceObject: {fileID: 0}
m_PrefabInstance: {fileID: 0}
m_PrefabAsset: {fileID: 0}
m_GameObject: {fileID: 7055079}
m_Enabled: 1
serializedVersion: 10
m_Type: 1
m_Shape: 0
m_Color: {r: 1, g: 0.95686275, b: 0.8392157, a: 1}
m_Intensity: 1.2
m_Range: 10
m_SpotAngle: 30
m_InnerSpotAngle: 21.80208
m_CookieSize: 10
m_Shadows:
  m_Type: 2
  m_Resolution: -1
  m_CustomResolution: -1
  m_Strength: 1
  m_Bias: 0.05
  m_NormalBias: 0.4
  m_NearPlane: 0.2
  m_CullingMatrixOverride:
    e00: 1
    e01: 0
    e02: 0
    e03: 0
    e10: 0
    e11: 1
    e12: 0
    e13: 0

```
      e20: 0
      e21: 0
      e22: 1
      e23: 0
      e30: 0
      e31: 0
      e32: 0
      e33: 1
    m_UseCullingMatrixOverride: 0
  m_Cookie: {fileID: 0}
  m_DrawHalo: 0
  m_Flare: {fileID: 0}
  m_RenderMode: 0
  m_CullingMask:
    serializedVersion: 2
    m_Bits: 4294967295
  m_RenderingLayerMask: 1
  m_Lightmapping: 1
  m_LightShadowCasterMode: 0
  m_AreaSize: {x: 1, y: 1}
  m_BounceIntensity: 1
  m_ColorTemperature: 6570
  m_UseColorTemperature: 0
  m_BoundingSphereOverride: {x: 0, y: 0, z: 0, w: 0}
  m_UseBoundingSphereOverride: 0
  m_UseViewFrustumForShadowCasterCull: 1
  m_ShadowRadius: 0
  m_ShadowAngle: 0
--- !u!4 &7055081
Transform:
  m_ObjectHideFlags: 0
```

```
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 7055079}
  m_LocalRotation: {x: 0.40821788, y: -0.23456968, z: 0.10938163, w:
0.8754261}
  m_LocalPosition: {x: 0, y: 100, z: 0}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 1
  m_LocalEulerAnglesHint: {x: 50, y: -30, z: 0}
--- !u!1 &963194225
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 963194228}
  - component: {fileID: 963194227}
  - component: {fileID: 963194226}
  - component: {fileID: 963194229}
  m_Layer: 0
  m_Name: PlayerCamera
  m_TagString: MainCamera
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
```

```
  m_IsActive: 1
--- !u!81 &963194226
AudioListener:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_Enabled: 1
--- !u!20 &963194227
Camera:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_Enabled: 1
  serializedVersion: 2
  m_ClearFlags: 1
  m_BackGroundColor: {r: 0.48, g: 0.69, b: 0.85, a: 1}
  m_projectionMatrixMode: 1
  m_GateFitMode: 2
  m_FOVAxisMode: 0
  m_SensorSize: {x: 36, y: 24}
  m_LensShift: {x: 0, y: 0}
  m_FocalLength: 50
  m_NormalizedViewPortRect:
    serializedVersion: 2
    x: 0
    y: 0
    width: 1
```

```
    height: 1
  m_near: 0.3
  m_far: 2000
  m_fieldOfView: 65
  m_orthographic: 0
  m_orthographicSize: 5
  m_Depth: -1
  m_CullingMask:
    serializedVersion: 2
    m_Bits: 4294967295
  m_RenderingPath: -1
  m_TargetTexture: {fileID: 0}
  m_TargetDisplay: 0
  m_TargetEye: 3
  m_HDR: 1
  m_AllowMSAA: 1
  m_AllowDynamicResolution: 0
  m_ForceIntoRT: 0
  m_OcclusionCulling: 1
  m_StereoConvergence: 10
  m_StereoSeparation: 0.022
--- !u!4 &963194228
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_LocalRotation: {x: 0.3420201, y: 0, z: 0, w: 0.9396927}
  m_LocalPosition: {x: 0, y: 15, z: -20}
  m_LocalScale: {x: 1, y: 1, z: 1}
```

```
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 0
  m_LocalEulerAnglesHint: {x: 40, y: 0, z: 0}
--- !u!114 &963194229
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 8b9a6c5d1c3e4b4a8f3e8b9c7d5e1f2a, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  target: {fileID: 0}
  offset: {x: 0, y: 15, z: -20}
  rotationSpeed: 2
  height: 15
  distance: 20
  isFixed: 1
  culturalCameraMode: 1
  prayerTimeCameraAdjustment: 1
  respectCulturalSites: 1
  smoothness: 5
  minHeight: 5
  maxHeight: 50
  minDistance: 10
```

```
  maxDistance: 100
--- !u!1 &1234567890
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 1234567891}
  - component: {fileID: 1234567892}
  - component: {fileID: 1234567893}
  - component: {fileID: 1234567894}
  - component: {fileID: 1234567895}
  m_Layer: 0
  m_Name: GameWorldManager
  m_TagString: GameController
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!4 &1234567891
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: 0, y: 0, z: 0}
  m_LocalScale: {x: 1, y: 1, z: 1}
```

```
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 2
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
--- !u!114 &1234567892
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 9c8b7d6e5f4a3c3b9e8f7e6d5c4b3a2, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  islandCount: 41
  currentIslandIndex: 0
  playerStartPosition: {x: 0, y: 0, z: 0}
  maldivianGeographicData: {fileID: 11400000, guid:
a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6, type: 2}
  gangManager: {fileID: 0}
  vehicleManager: {fileID: 0}
  buildingManager: {fileID: 0}
  floraManager: {fileID: 0}
  weatherSystem: {fileID: 0}
  prayerTimeSystem: {fileID: 0}
  islamicCalendar: {fileID: 0}
  boduberuSystem: {fileID: 0}
```

```
  fishingSystem: {fileID: 0}
  culturalEventManager: {fileID: 0}
  missionManager: {fileID: 0}
  saveSystem: {fileID: 0}
  audioManager: {fileID: 0}
  inputManager: {fileID: 0}
  uiManager: {fileID: 0}
--- !u!114 &1234567893
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 8d7c6b5e4f3a2b1a9e8f7e6d5c4b3a2, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  oceanMaterial: {fileID: 2100000, guid: b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7,
type: 2}
  beachMaterial: {fileID: 2100000, guid: c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8,
type: 2}
  palmTreePrefab: {fileID: 100000, guid: d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9,
type: 3}
  coconutTreePrefab: {fileID: 100000, guid: e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0,
type: 3}
  coralReefPrefab: {fileID: 100000, guid: f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0u1,
type: 3}
```

lagoonPrefab: {fileID: 100000, guid: g7h8i9j0k1l2m3n4o5p6q7r8s9t0u1v2, type: 3}
harborPrefab: {fileID: 100000, guid: h8i9j0k1l2m3n4o5p6q7r8s9t0u1v2w3, type: 3}
airportPrefab: {fileID: 100000, guid: i9j0k1l2m3n4o5p6q7r8s9t0u1v2w3x4, type: 3}
resortPrefab: {fileID: 100000, guid: j0k1l2m3n4o5p6q7r8s9t0u1v2w3x4y5, type: 3}
governmentBuildingPrefab: {fileID: 100000, guid: k1l2m3n4o5p6q7r8s9t0u1v2w3x4y5z6, type: 3}
schoolPrefab: {fileID: 100000, guid: l2m3n4o5p6q7r8s9t0u1v2w3x4y5z6a7, type: 3}
hospitalPrefab: {fileID: 100000, guid: m3n4o5p6q7r8s9t0u1v2w3x4y5z6a7b8, type: 3}
shopPrefab: {fileID: 100000, guid: n4o5p6q7r8s9t0u1v2w3x4y5z6a7b8c9, type: 3}
restaurantPrefab: {fileID: 100000, guid: o5p6q7r8s9t0u1v2w3x4y5z6a7b8c9d0, type: 3}
cafePrefab: {fileID: 100000, guid: p6q7r8s9t0u1v2w3x4y5z6a7b8c9d0e1, type: 3}
guestHousePrefab: {fileID: 100000, guid: q7r8s9t0u1v2w3x4y5z6a7b8c9d0e1f2, type: 3}
localHousePrefab: {fileID: 100000, guid: r8s9t0u1v2w3x4y5z6a7b8c9d0e1f2g3, type: 3}
historicalSitePrefab: {fileID: 100000, guid: s9t0u1v2w3x4y5z6a7b8c9d0e1f2g3h4, type: 3}
ancientRuinsPrefab: {fileID: 100000, guid: t0u1v2w3x4y5z6a7b8c9d0e1f2g3h4i5, type: 3}
traditionalWellPrefab: {fileID: 100000, guid: u1v2w3x4y5z6a7b8c9d0e1f2g3h4i5j6, type: 3}

communityCenterPrefab: {fileID: 100000, guid: v2w3x4y5z6a7b8c9d0e1f2g3h4i5j6k7, type: 3}
sportsFieldPrefab: {fileID: 100000, guid: w3x4y5z6a7b8c9d0e1f2g3h4i5j6k7l8, type: 3}
playgroundPrefab: {fileID: 100000, guid: x4y5z6a7b8c9d0e1f2g3h4i5j6k7l8m9, type: 3}
parkPrefab: {fileID: 100000, guid: y5z6a7b8c9d0e1f2g3h4i5j6k7l8m9n0, type: 3}
gardenPrefab: {fileID: 100000, guid: z6a7b8c9d0e1f2g3h4i5j6k7l8m9n0o1, type: 3}
farmLandPrefab: {fileID: 100000, guid: a7b8c9d0e1f2g3h4i5j6k7l8m9n0o1p2, type: 3}
mosquePrefab: {fileID: 100000, guid: b8c9d0e1f2g3h4i5j6k7l8m9n0o1p2q3, type: 3}
islamicSymbolPrefab: {fileID: 100000, guid: c9d0e1f2g3h4i5j6k7l8m9n0o1p2q3r4, type: 3}
maldivianFlagPrefab: {fileID: 100000, guid: d0e1f2g3h4i5j6k7l8m9n0o1p2q3r4s5, type: 3}
traditionalDressPrefab: {fileID: 100000, guid: e1f2g3h4i5j6k7l8m9n0o1p2q3r4s5t6, type: 3}
localCraftPrefab: {fileID: 100000, guid: f2g3h4i5j6k7l8m9n0o1p2q3r4s5t6u7, type: 3}
fishingBoatPrefab: {fileID: 100000, guid: g3h4i5j6k7l8m9n0o1p2q3r4s5t6u7v8, type: 3}
speedBoatPrefab: {fileID: 100000, guid: h4i5j6k7l8m9n0o1p2q3r4s5t6u7v8w9, type: 3}
yachtPrefab: {fileID: 100000, guid: i5j6k7l8m9n0o1p2q3r4s5t6u7v8w9x0, type: 3}
seaplanePrefab: {fileID: 100000, guid: j6k7l8m9n0o1p2q3r4s5t6u7v8w9x0y1, type: 3}
jetSkiPrefab: {fileID: 100000, guid: k7l8m9n0o1p2q3r4s5t6u7v8w9x0y1z2, type: 3}

traditionalBoatPrefab: {fileID: 100000, guid:
l8m9n0o1p2q3r4s5t6u7v8w9x0y1z2a3, type: 3}
dhoniPrefab: {fileID: 100000, guid: m9n0o1p2q3r4s5t6u7v8w9x0y1z2a3b4,
type: 3}
bancaPrefab: {fileID: 100000, guid: n0o1p2q3r4s5t6u7v8w9x0y1z2a3b4c5,
type: 3}
fishingNetPrefab: {fileID: 100000, guid: o1p2q3r4s5t6u7v8w9x0y1z2a3b4c5d6,
type: 3}
anchorPrefab: {fileID: 100000, guid: p2q3r4s5t6u7v8w9x0y1z2a3b4c5d6e7,
type: 3}
lighthousePrefab: {fileID: 100000, guid: q3r4s5t6u7v8w9x0y1z2a3b4c5d6e7f8,
type: 3}
buoyPrefab: {fileID: 100000, guid: r4s5t6u7v8w9x0y1z2a3b4c5d6e7f8g9, type:
3}
wavePrefab: {fileID: 100000, guid: s5t6u7v8w9x0y1z2a3b4c5d6e7f8g9h0, type:
3}
currentPrefab: {fileID: 100000, guid: t6u7v8w9x0y1z2a3b4c5d6e7f8g9h0i1,
type: 3}
tidePrefab: {fileID: 100000, guid: u7v8w9x0y1z2a3b4c5d6e7f8g9h0i1j2, type: 3}
windZonePrefab: {fileID: 100000, guid: v8w9x0y1z2a3b4c5d6e7f8g9h0i1j2k3,
type: 3}
rainZonePrefab: {fileID: 100000, guid: w9x0y1z2a3b4c5d6e7f8g9h0i1j2k3l4,
type: 3}
cloudPrefab: {fileID: 100000, guid: x0y1z2a3b4c5d6e7f8g9h0i1j2k3l4m5, type:
3}
sunPrefab: {fileID: 100000, guid: y1z2a3b4c5d6e7f8g9h0i1j2k3l4m5n6, type: 3}
moonPrefab: {fileID: 100000, guid: z2a3b4c5d6e7f8g9h0i1j2k3l4m5n6o7, type:
3}
starPrefab: {fileID: 100000, guid: a3b4c5d6e7f8g9h0i1j2k3l4m5n6o7p8, type: 3}
prayerCallPrefab: {fileID: 100000, guid: b4c5d6e7f8g9h0i1j2k3l4m5n6o7p8q9,
type: 3}

```
  azanPrefab: {fileID: 100000, guid: c5d6e7f8g9h0i1j2k3l4m5n6o7p8q9r0, type:
3}
  mosqueBellPrefab: {fileID: 100000, guid: d6e7f8g9h0i1j2k3l4m5n6o7p8q9r0s1,
type: 3}
  traditionalAnnouncementPrefab: {fileID: 100000, guid:
e7f8g9h0i1j2k3l4m5n6o7p8q9r0s1t2, type: 3}
  communityDrumPrefab: {fileID: 100000, guid:
f8g9h0i1j2k3l4m5n6o7p8q9r0s1t2u3, type: 3}
  celebrationMusicPrefab: {fileID: 100000, guid:
g9h0i1j2k3l4m5n6o7p8q9r0s1t2u3v4, type: 3}
  weddingMusicPrefab: {fileID: 100000, guid:
h0i1j2k3l4m5n6o7p8q9r0s1t2u3v4w5, type: 3}
  funeralBellPrefab: {fileID: 100000, guid: i1j2k3l4m5n6o7p8q9r0s1t2u3v4w5x6,
type: 3}
  respectfulSilencePrefab: {fileID: 100000, guid:
j2k3l4m5n6o7p8q9r0s1t2u3v4w5x6y7, type: 3}
--- !u!114 &1234567894
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 9e8d7c6b5e4f3a2b1a9e8f7e6d5c4b3, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  gameState: 0
  currentMission: {fileID: 0}
```

```
  playerProgress: {fileID: 0}
  culturalSettings:
    respectPrayerTimes: 1
    preserveCulturalFidelity: 1
    reducedVisualIntensity: 0
    disableLoudEffects: 0
    simplifiedUI: 0
  maldivianSettings:
    useRealGeographicData: 1
    enableTraditionalEvents: 1
    respectIslamicValues: 1
    preserveLocalCulture: 1
    communityFriendlyInteractions: 1
  weatherSettings:
    enableMonsoonSimulation: 1
    realisticOceanCurrents: 1
    traditionalWeatherPatterns: 1
    culturalWeatherEvents: 1
  audioSettings:
    respectPrayerTimeAudio: 1
    traditionalMusicIntegration: 1
    culturalSoundEffects: 1
    communityFriendlyAudio: 1
--- !u!114 &1234567895
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
```

```
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 8f7e6d5c4b3a2a1a9e8f7e6d5c4b3a2, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  worldName: "Maldivian Archipelago"
  worldSize: {x: 10000, y: 1000, z: 10000}
  islandDistribution: 0
  oceanDepth: -50
  beachWidth: 20
  culturalSiteDensity: 0.8
  religiousSiteDensity: 0.6
  communityAreaDensity: 0.7
  traditionalZoneDensity: 0.9
  spawnPointCount: 83
  savePointCount: 41
  checkpointCount: 200
  missionObjectiveCount: 500
  collectibleCount: 1000
  weatherSystem: {fileID: 0}
  timeManager: {fileID: 0}
  dayNightCycle: 1
  realisticTides: 1
  monsoonEffects: 1
  culturalEvents: 1
  islamicCalendar: 1
  prayerTimes: 1
  traditionalActivities: 1
  communityEvents: 1
```

64. LoadingScreen.unity

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!29 &1
OcclusionCullingSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 2
  m_OcclusionBakeSettings:
    smallestOccluder: 5
    smallestHole: 0.25
    backfaceThreshold: 100
  m_SceneGUID: 00000000000000000000000000000000
  m_OcclusionCullingData: {fileID: 0}
--- !u!104 &2
RenderSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 9
  m_Fog: 0
  m_FogColor: {r: 0.5, g: 0.5, b: 0.5, a: 1}
  m_FogMode: 3
  m_FogDensity: 0.01
  m_LinearFogStart: 0
  m_LinearFogEnd: 300
  m_AmbientSkyColor: {r: 0.212, g: 0.227, b: 0.259, a: 1}
  m_AmbientEquatorColor: {r: 0.114, g: 0.125, b: 0.133, a: 1}
  m_AmbientGroundColor: {r: 0.047, g: 0.043, b: 0.035, a: 1}
  m_AmbientIntensity: 1
```

```
  m_AmbientMode: 0
  m_SubtractiveShadowColor: {r: 0.42, g: 0.478, b: 0.627, a: 1}
  m_SkyboxMaterial: {fileID: 10304, guid:
000000000000000f0000000000000000, type: 0}
  m_HaloStrength: 0.5
  m_FlareStrength: 1
  m_FlareFadeSpeed: 3
  m_HaloTexture: {fileID: 0}
  m_SpotCookie: {fileID: 10001, guid: 0000000000000000e000000000000000,
type: 0}
  m_DefaultReflectionMode: 0
  m_DefaultReflectionResolution: 128
  m_ReflectionBounces: 1
  m_ReflectionIntensity: 1
  m_CustomReflection: {fileID: 0}
  m_Sun: {fileID: 0}
  m_IndirectSpecularColor: {r: 0.44657898, g: 0.4964133, b: 0.5748178, a: 1}
  m_UseRadianceAmbientProbe: 0
--- !u!157 &3
LightmapSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 12
  m_GIWorkflowMode: 1
  m_GISettings:
    serializedVersion: 2
    m_BounceScale: 1
    m_IndirectOutputScale: 1
    m_AlbedoBoost: 1
    m_EnvironmentLightingMode: 0
    m_EnableBakedLightmaps: 1
    m_EnableRealtimeLightmaps: 0
```

```
m_LightmapEditorSettings:
  serializedVersion: 12
  m_Resolution: 2
  m_BakeResolution: 40
  m_AtlasSize: 1024
  m_AO: 0
  m_AOMaxDistance: 1
  m_CompAOExponent: 1
  m_CompAOExponentDirect: 0
  m_ExtractAmbientOcclusion: 0
  m_Padding: 2
  m_LightmapParameters: {fileID: 0}
  m_LightmapsBakeMode: 1
  m_TextureCompression: 1
  m_FinalGather: 0
  m_FinalGatherFiltering: 1
  m_FinalGatherRayCount: 256
  m_ReflectionCompression: 2
  m_MixedBakeMode: 2
  m_BakeBackend: 1
  m_PVRSampling: 1
  m_PVRDirectSampleCount: 32
  m_PVRSampleCount: 512
  m_PVRBounces: 2
  m_PVREnvironmentSampleCount: 256
  m_PVREnvironmentReferencePointCount: 2048
  m_PVRFilteringMode: 1
  m_PVRDenoiserTypeDirect: 1
  m_PVRDenoiserTypeIndirect: 1
  m_PVRDenoiserTypeAO: 1
  m_PVRFilterTypeDirect: 0
```

```
    m_PVRFilterTypeIndirect: 0
    m_PVRFilterTypeAO: 0
    m_PVREnvironmentMIS: 1
    m_PVRCulling: 1
    m_PVRFilteringGaussRadiusDirect: 1
    m_PVRFilteringGaussRadiusIndirect: 5
    m_PVRFilteringAtrousPositionSigmaDirect: 0.5
    m_PVRFilteringAtrousPositionSigmaIndirect: 2
    m_PVRFilteringAtrousPositionSigmaAO: 1
    m_ExportTrainingData: 0
    m_TrainingDataDestination: TrainingData
    m_LightProbeSampleCountMultiplier: 4
  m_LightingDataAsset: {fileID: 0}
  m_LightingSettings: {fileID: 0}
--- !u!196 &4
NavMeshSettings:
  serializedVersion: 2
  m_ObjectHideFlags: 0
  m_BuildSettings:
    serializedVersion: 2
    agentTypeID: 0
    agentRadius: 0.5
    agentHeight: 2
    agentSlope: 45
    agentClimb: 0.4
    ledgeDropHeight: 0
    maxJumpAcrossDistance: 0
    minRegionArea: 2
    manualCellSize: 0
    cellSize: 0.16666667
    manualTileSize: 0
```

```
    tileSize: 256
    accuratePlacement: 0
    maxJobWorkers: 0
    preserveTilesOutsideBounds: 0
    debug:
      m_Flags: 0
  m_NavMeshData: {fileID: 0}
--- !u!1 &7055079
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 7055081}
  - component: {fileID: 7055080}
  m_Layer: 0
  m_Name: Directional Light
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!108 &7055080
Light:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 7055079}
```

m_Enabled: 1
serializedVersion: 10
m_Type: 1
m_Shape: 0
m_Color: {r: 1, g: 0.95686275, b: 0.8392157, a: 1}
m_Intensity: 1
m_Range: 10
m_SpotAngle: 30
m_InnerSpotAngle: 21.80208
m_CookieSize: 10
m_Shadows:
  m_Type: 2
  m_Resolution: -1
  m_CustomResolution: -1
  m_Strength: 1
  m_Bias: 0.05
  m_NormalBias: 0.4
  m_NearPlane: 0.2
  m_CullingMatrixOverride:
    e00: 1
    e01: 0
    e02: 0
    e03: 0
    e10: 0
    e11: 1
    e12: 0
    e13: 0
    e20: 0
    e21: 0
    e22: 1
    e23: 0

```
      e30: 0
      e31: 0
      e32: 0
      e33: 1
    m_UseCullingMatrixOverride: 0
  m_Cookie: {fileID: 0}
  m_DrawHalo: 0
  m_Flare: {fileID: 0}
  m_RenderMode: 0
  m_CullingMask:
    serializedVersion: 2
    m_Bits: 4294967295
  m_RenderingLayerMask: 1
  m_Lightmapping: 1
  m_LightShadowCasterMode: 0
  m_AreaSize: {x: 1, y: 1}
  m_BounceIntensity: 1
  m_ColorTemperature: 6570
  m_UseColorTemperature: 0
  m_BoundingSphereOverride: {x: 0, y: 0, z: 0, w: 0}
  m_UseBoundingSphereOverride: 0
  m_UseViewFrustumForShadowCasterCull: 1
  m_ShadowRadius: 0
  m_ShadowAngle: 0
--- !u!4 &7055081
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 7055079}
```

```
  m_LocalRotation: {x: 0.40821788, y: -0.23456968, z: 0.10938163, w:
0.8754261}
  m_LocalPosition: {x: 0, y: 3, z: 0}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 1
  m_LocalEulerAnglesHint: {x: 50, y: -30, z: 0}
--- !u!1 &963194225
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 963194228}
  - component: {fileID: 963194227}
  - component: {fileID: 963194226}
  m_Layer: 0
  m_Name: LoadingCamera
  m_TagString: MainCamera
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!81 &963194226
AudioListener:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
```

```
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_Enabled: 1
--- !u!20 &963194227
Camera:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_Enabled: 1
  serializedVersion: 2
  m_ClearFlags: 1
  m_BackGroundColor: {r: 0.13725491, g: 0.12156863, b: 0.1254902, a: 1}
  m_projectionMatrixMode: 1
  m_GateFitMode: 2
  m_FOVAxisMode: 0
  m_SensorSize: {x: 36, y: 24}
  m_LensShift: {x: 0, y: 0}
  m_FocalLength: 50
  m_NormalizedViewPortRect:
    serializedVersion: 2
    x: 0
    y: 0
    width: 1
    height: 1
  m_near: 0.3
  m_far: 1000
  m_fieldOfView: 60
  m_orthographic: 0
```

```
  m_orthographicSize: 5
  m_Depth: -1
  m_CullingMask:
    serializedVersion: 2
    m_Bits: 4294967295
  m_RenderingPath: -1
  m_TargetTexture: {fileID: 0}
  m_TargetDisplay: 0
  m_TargetEye: 3
  m_HDR: 1
  m_AllowMSAA: 1
  m_AllowDynamicResolution: 0
  m_ForceIntoRT: 0
  m_OcclusionCulling: 1
  m_StereoConvergence: 10
  m_StereoSeparation: 0.022
--- !u!4 &963194228
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: 0, y: 1, z: -10}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 0
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
```

```
--- !u!1 &1234567890
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 1234567891}
  - component: {fileID: 1234567892}
  - component: {fileID: 1234567893}
  m_Layer: 5
  m_Name: LoadingUI
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!224 &1234567891
RectTransform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: 0, y: 0, z: 0}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
```

```
  m_RootOrder: 2
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
  m_AnchorMin: {x: 0.5, y: 0.5}
  m_AnchorMax: {x: 0.5, y: 0.5}
  m_AnchoredPosition: {x: 0, y: 0}
  m_SizeDelta: {x: 100, y: 100}
  m_Pivot: {x: 0.5, y: 0.5}
--- !u!114 &1234567892
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 7b8c9d6e5f4a3c3b9e8f7e6d5c4b3a2, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  loadingText: "Loading Maldivian World..."
  progressBar: {fileID: 0}
  progressText: {fileID: 0}
  loadingTips:
  - "The Maldives consists of 1,192 coral islands grouped in 26 atolls"
  - "Islam is the official religion and plays a central role in daily life"
  - "Traditional Maldivian music includes Boduberu drumming performances"
  - "Fishing is a major industry and part of cultural heritage"
  - "Respect prayer times and religious observances"
  - "The traditional Maldivian boat is called a 'dhoni'"
  - "Coconut palms are integral to Maldivian culture and economy"
```

```
  - "Community harmony and respect are highly valued"
  culturalTips:
  - "Always remove shoes before entering homes or mosques"
  - "Dress modestly, especially when visiting local islands"
  - "Friday is the holy day - many businesses close for prayers"
  - "Ramadan is observed with fasting from dawn to sunset"
  - "Traditional crafts include mat weaving and lacquer work"
  - "Boduberu music brings communities together for celebrations"
  - "Fishing competitions are popular community events"
  - "Respect for elders and community leaders is important"
  backgroundVideoPlayer: {fileID: 0}
  traditionalMusicPlayer: {fileID: 0}
  ambientOceanSounds: {fileID: 0}
  loadingCanvas: {fileID: 0}
  tipDisplayText: {fileID: 0}
  culturalTipDisplayText: {fileID: 0}
  loadingAnimation: {fileID: 0}
  minLoadingTime: 3
  maxLoadingTime: 10
  fadeInDuration: 1
  fadeOutDuration: 1
  respectPrayerTimes: 1
  culturalSensitivityMode: 1
--- !u!114 &1234567893
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
```

```
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 6c7b8d9e8f7a6b5c4d3e2f1a9e8d7c6, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  scenesToLoad:
  - GameWorld
  - MainMenu
  - OptionsMenu
  loadingPriority: 0
  allowSceneActivation: 1
  progressThreshold: 0.9
  culturalLoadingMode: 1
  prayerTimeAwareLoading: 1
  communityFriendlyLoading: 1
  traditionalMusicDuringLoad: 1
  respectCulturalSettings: 1
```

65. OptionsMenu.unity

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!29 &1
OcclusionCullingSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 2
  m_OcclusionBakeSettings:
    smallestOccluder: 5
```

```
    smallestHole: 0.25
    backfaceThreshold: 100
  m_SceneGUID: 00000000000000000000000000000000
  m_OcclusionCullingData: {fileID: 0}
--- !u!104 &2
RenderSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 9
  m_Fog: 0
  m_FogColor: {r: 0.5, g: 0.5, b: 0.5, a: 1}
  m_FogMode: 3
  m_FogDensity: 0.01
  m_LinearFogStart: 0
  m_LinearFogEnd: 300
  m_AmbientSkyColor: {r: 0.212, g: 0.227, b: 0.259, a: 1}
  m_AmbientEquatorColor: {r: 0.114, g: 0.125, b: 0.133, a: 1}
  m_AmbientGroundColor: {r: 0.047, g: 0.043, b: 0.035, a: 1}
  m_AmbientIntensity: 1
  m_AmbientMode: 0
  m_SubtractiveShadowColor: {r: 0.42, g: 0.478, b: 0.627, a: 1}
  m_SkyboxMaterial: {fileID: 10304, guid:
000000000000000f0000000000000000, type: 0}
  m_HaloStrength: 0.5
  m_FlareStrength: 1
  m_FlareFadeSpeed: 3
  m_HaloTexture: {fileID: 0}
  m_SpotCookie: {fileID: 10001, guid: 0000000000000000e000000000000000,
type: 0}
  m_DefaultReflectionMode: 0
  m_DefaultReflectionResolution: 128
  m_ReflectionBounces: 1
```

```
  m_ReflectionIntensity: 1
  m_CustomReflection: {fileID: 0}
  m_Sun: {fileID: 0}
  m_IndirectSpecularColor: {r: 0.44657898, g: 0.4964133, b: 0.5748178, a: 1}
  m_UseRadianceAmbientProbe: 0
--- !u!157 &3
LightmapSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 12
  m_GIWorkflowMode: 1
  m_GISettings:
    serializedVersion: 2
    m_BounceScale: 1
    m_IndirectOutputScale: 1
    m_AlbedoBoost: 1
    m_EnvironmentLightingMode: 0
    m_EnableBakedLightmaps: 1
    m_EnableRealtimeLightmaps: 0
  m_LightmapEditorSettings:
    serializedVersion: 12
    m_Resolution: 2
    m_BakeResolution: 40
    m_AtlasSize: 1024
    m_AO: 0
    m_AOMaxDistance: 1
    m_CompAOExponent: 1
    m_CompAOExponentDirect: 0
    m_ExtractAmbientOcclusion: 0
    m_Padding: 2
    m_LightmapParameters: {fileID: 0}
    m_LightmapsBakeMode: 1
```

```
  m_TextureCompression: 1
  m_FinalGather: 0
  m_FinalGatherFiltering: 1
  m_FinalGatherRayCount: 256
  m_ReflectionCompression: 2
  m_MixedBakeMode: 2
  m_BakeBackend: 1
  m_PVRSampling: 1
  m_PVRDirectSampleCount: 32
  m_PVRSampleCount: 512
  m_PVRBounces: 2
  m_PVREnvironmentSampleCount: 256
  m_PVREnvironmentReferencePointCount: 2048
  m_PVRFilteringMode: 1
  m_PVRDenoiserTypeDirect: 1
  m_PVRDenoiserTypeIndirect: 1
  m_PVRDenoiserTypeAO: 1
  m_PVRFilterTypeDirect: 0
  m_PVRFilterTypeIndirect: 0
  m_PVRFilterTypeAO: 0
  m_PVREnvironmentMIS: 1
  m_PVRCulling: 1
  m_PVRFilteringGaussRadiusDirect: 1
  m_PVRFilteringGaussRadiusIndirect: 5
  m_PVRFilteringAtrousPositionSigmaDirect: 0.5
  m_PVRFilteringAtrousPositionSigmaIndirect: 2
  m_PVRFilteringAtrousPositionSigmaAO: 1
  m_ExportTrainingData: 0
  m_TrainingDataDestination: TrainingData
  m_LightProbeSampleCountMultiplier: 4
m_LightingDataAsset: {fileID: 0}
```

```
  m_LightingSettings: {fileID: 0}
--- !u!196 &4
NavMeshSettings:
  serializedVersion: 2
  m_ObjectHideFlags: 0
  m_BuildSettings:
    serializedVersion: 2
    agentTypeID: 0
    agentRadius: 0.5
    agentHeight: 2
    agentSlope: 45
    agentClimb: 0.4
    ledgeDropHeight: 0
    maxJumpAcrossDistance: 0
    minRegionArea: 2
    manualCellSize: 0
    cellSize: 0.16666667
    manualTileSize: 0
    tileSize: 256
    accuratePlacement: 0
    maxJobWorkers: 0
    preserveTilesOutsideBounds: 0
    debug:
      m_Flags: 0
  m_NavMeshData: {fileID: 0}
--- !u!1 &7055079
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
```

```
  serializedVersion: 6
  m_Component:
  - component: {fileID: 7055081}
  - component: {fileID: 7055080}
  m_Layer: 0
  m_Name: Directional Light
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!108 &7055080
Light:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 7055079}
  m_Enabled: 1
  serializedVersion: 10
  m_Type: 1
  m_Shape: 0
  m_Color: {r: 1, g: 0.95686275, b: 0.8392157, a: 1}
  m_Intensity: 1
  m_Range: 10
  m_SpotAngle: 30
  m_InnerSpotAngle: 21.80208
  m_CookieSize: 10
  m_Shadows:
    m_Type: 2
    m_Resolution: -1
```

```
  m_CustomResolution: -1
  m_Strength: 1
  m_Bias: 0.05
  m_NormalBias: 0.4
  m_NearPlane: 0.2
  m_CullingMatrixOverride:
    e00: 1
    e01: 0
    e02: 0
    e03: 0
    e10: 0
    e11: 1
    e12: 0
    e13: 0
    e20: 0
    e21: 0
    e22: 1
    e23: 0
    e30: 0
    e31: 0
    e32: 0
    e33: 1
  m_UseCullingMatrixOverride: 0
m_Cookie: {fileID: 0}
m_DrawHalo: 0
m_Flare: {fileID: 0}
m_RenderMode: 0
m_CullingMask:
  serializedVersion: 2
  m_Bits: 4294967295
m_RenderingLayerMask: 1
```

```
  m_Lightmapping: 1
  m_LightShadowCasterMode: 0
  m_AreaSize: {x: 1, y: 1}
  m_BounceIntensity: 1
  m_ColorTemperature: 6570
  m_UseColorTemperature: 0
  m_BoundingSphereOverride: {x: 0, y: 0, z: 0, w: 0}
  m_UseBoundingSphereOverride: 0
  m_UseViewFrustumForShadowCasterCull: 1
  m_ShadowRadius: 0
  m_ShadowAngle: 0
--- !u!4 &7055081
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 7055079}
  m_LocalRotation: {x: 0.40821788, y: -0.23456968, z: 0.10938163, w:
0.8754261}
  m_LocalPosition: {x: 0, y: 3, z: 0}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 1
  m_LocalEulerAnglesHint: {x: 50, y: -30, z: 0}
--- !u!1 &963194225
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
```

```
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 963194228}
  - component: {fileID: 963194227}
  - component: {fileID: 963194226}
  m_Layer: 0
  m_Name: OptionsCamera
  m_TagString: MainCamera
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!81 &963194226
AudioListener:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_Enabled: 1
--- !u!20 &963194227
Camera:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_Enabled: 1
  serializedVersion: 2
```

m_ClearFlags: 1
m_BackGroundColor: {r: 0.19215687, g: 0.3019608, b: 0.4745098, a: 0}
m_projectionMatrixMode: 1
m_GateFitMode: 2
m_FOVAxisMode: 0
m_SensorSize: {x: 36, y: 24}
m_LensShift: {x: 0, y: 0}
m_FocalLength: 50
m_NormalizedViewPortRect:
  serializedVersion: 2
  x: 0
  y: 0
  width: 1
  height: 1
m_near: 0.3
m_far: 1000
m_fieldOfView: 60
m_orthographic: 0
m_orthographicSize: 5
m_Depth: -1
m_CullingMask:
  serializedVersion: 2
  m_Bits: 4294967295
m_RenderingPath: -1
m_TargetTexture: {fileID: 0}
m_TargetDisplay: 0
m_TargetEye: 3
m_HDR: 1
m_AllowMSAA: 1
m_AllowDynamicResolution: 0
m_ForceIntoRT: 0

```
  m_OcclusionCulling: 1
  m_StereoConvergence: 10
  m_StereoSeparation: 0.022
--- !u!4 &963194228
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 963194225}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: 0, y: 1, z: -10}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 0
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
--- !u!1 &1234567890
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 1234567891}
  - component: {fileID: 1234567892}
  - component: {fileID: 1234567893}
  - component: {fileID: 1234567894}
  - component: {fileID: 1234567895}
```

```
  m_Layer: 5
  m_Name: OptionsUI
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!224 &1234567891
RectTransform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: 0, y: 0, z: 0}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 2
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
  m_AnchorMin: {x: 0.5, y: 0.5}
  m_AnchorMax: {x: 0.5, y: 0.5}
  m_AnchoredPosition: {x: 0, y: 0}
  m_SizeDelta: {x: 100, y: 100}
  m_Pivot: {x: 0.5, y: 0.5}
--- !u!114 &1234567892
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
```

```
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 9c8d7e6f5a4b3c2d1e0f9a8b7c6d5e4, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  optionsTitle: "Game Settings"
  graphicsTabText: "Graphics"
  audioTabText: "Audio"
  gameplayTabText: "Gameplay"
  culturalTabText: "Cultural"
  controlsTabText: "Controls"
  languageTabText: "Language"
  backButtonText: "Back"
  applyButtonText: "Apply"
  resetButtonText: "Reset to Defaults"
  graphicsSettings:
    resolutionDropdown: {fileID: 0}
    qualityDropdown: {fileID: 0}
    fullscreenToggle: {fileID: 0}
    vsyncToggle: {fileID: 0}
    antiAliasingDropdown: {fileID: 0}
    textureQualityDropdown: {fileID: 0}
    shadowQualityDropdown: {fileID: 0}
    viewDistanceSlider: {fileID: 0}
    motionBlurToggle: {fileID: 0}
    bloomToggle: {fileID: 0}
  audioSettings:
```

masterVolumeSlider: {fileID: 0}
musicVolumeSlider: {fileID: 0}
sfxVolumeSlider: {fileID: 0}
ambientVolumeSlider: {fileID: 0}
traditionalMusicVolumeSlider: {fileID: 0}
prayerCallVolumeSlider: {fileID: 0}
audioOutputDropdown: {fileID: 0}
subtitleToggle: {fileID: 0}
subtitleLanguageDropdown: {fileID: 0}
respectPrayerTimesToggle: {fileID: 0}
culturalAudioModeDropdown: {fileID: 0}
gameplaySettings:
difficultyDropdown: {fileID: 0}
gameSpeedSlider: {fileID: 0}
autoSaveIntervalSlider: {fileID: 0}
tutorialToggle: {fileID: 0}
hintsToggle: {fileID: 0}
objectiveMarkersToggle: {fileID: 0}
minimapToggle: {fileID: 0}
waypointToggle: {fileID: 0}
culturalInteractionModeDropdown: {fileID: 0}
respectCommunityValuesToggle: {fileID: 0}
traditionalActivityPromptsToggle: {fileID: 0}
culturalSettings:
respectPrayerTimesToggle: {fileID: 0}
preserveCulturalFidelityToggle: {fileID: 0}
reducedVisualIntensityToggle: {fileID: 0}
disableLoudEffectsToggle: {fileID: 0}
simplifiedUIToggle: {fileID: 0}
culturalSensitivityModeDropdown: {fileID: 0}
traditionalEventNotificationsToggle: {fileID: 0}

```
  islamicCalendarDisplayToggle: {fileID: 0}
  maldivianDateFormatToggle: {fileID: 0}
  communityFriendlyInteractionsToggle: {fileID: 0}
  respectReligiousSitesToggle: {fileID: 0}
  preserveLocalCultureToggle: {fileID: 0}
  traditionalMusicIntegrationToggle: {fileID: 0}
  culturalLearningModeToggle: {fileID: 0}
  environmentalAwarenessToggle: {fileID: 0}
  marineConservationAwarenessToggle: {fileID: 0}
controlSettings:
  inputMethodDropdown: {fileID: 0}
  mouseSensitivitySlider: {fileID: 0}
  gamepadSensitivitySlider: {fileID: 0}
  touchSensitivitySlider: {fileID: 0}
  invertYAxisToggle: {fileID: 0}
  vibrationToggle: {fileID: 0}
  hapticFeedbackToggle: {fileID: 0}
  keyBindingButtons: []
  resetControlsButton: {fileID: 0}
  culturalGestureModeDropdown: {fileID: 0}
  traditionalInputMethodsToggle: {fileID: 0}
languageSettings:
  languageDropdown: {fileID: 0}
  subtitleLanguageDropdown: {fileID: 0}
  audioLanguageDropdown: {fileID: 0}
  culturalTermsModeDropdown: {fileID: 0}
  traditionalGreetingsToggle: {fileID: 0}
  dhivehiLanguageSupportToggle: {fileID: 0}
  arabicScriptSupportToggle: {fileID: 0}
  culturalContextTooltipsToggle: {fileID: 0}
optionsCanvas: {fileID: 0}
```

```
  tabButtons: []
  settingsPanels: []
  backButton: {fileID: 0}
  applyButton: {fileID: 0}
  resetButton: {fileID: 0}
  currentTab: 0
  hasUnsavedChanges: 0
  culturalMode: 1
  respectPrayerTimes: 1
  communityFriendlyDefaults: 1
--- !u!114 &1234567893
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 8d7c6b5e4f3a2b1a9e8f7e6d5c4b3a2, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  settingsVersion: "1.0.0"
  defaultSettings:
    graphicsQuality: 2
    resolution: {x: 1920, y: 1080}
    fullscreen: 1
    vsync: 1
    antiAliasing: 2
    textureQuality: 1
```

shadowQuality: 2
viewDistance: 1000
motionBlur: 1
bloom: 1
masterVolume: 0.8
musicVolume: 0.7
sfxVolume: 0.9
ambientVolume: 0.6
traditionalMusicVolume: 0.8
prayerCallVolume: 0.5
respectPrayerTimes: 1
culturalAudioMode: 1
difficulty: 1
gameSpeed: 1
autoSaveInterval: 300
tutorial: 1
hints: 1
objectiveMarkers: 1
minimap: 1
waypoint: 1
culturalInteractionMode: 1
respectCommunityValues: 1
traditionalActivityPrompts: 1
respectPrayerTimes: 1
preserveCulturalFidelity: 1
reducedVisualIntensity: 0
disableLoudEffects: 0
simplifiedUI: 0
culturalSensitivityMode: 1
traditionalEventNotifications: 1
islamicCalendarDisplay: 1

```
  maldivianDateFormat: 1
  communityFriendlyInteractions: 1
  respectReligiousSites: 1
  preserveLocalCulture: 1
  traditionalMusicIntegration: 1
  culturalLearningMode: 1
  environmentalAwareness: 1
  marineConservationAwareness: 1
  inputMethod: 0
  mouseSensitivity: 1
  gamepadSensitivity: 1
  touchSensitivity: 1
  invertYAxis: 0
  vibration: 1
  hapticFeedback: 1
  culturalGestureMode: 1
  traditionalInputMethods: 1
  language: 0
  subtitleLanguage: 0
  audioLanguage: 0
  culturalTermsMode: 1
  traditionalGreetings: 1
  dhivehiLanguageSupport: 1
  arabicScriptSupport: 1
  culturalContextTooltips: 1
currentSettings: {}
settingsFilePath: "RVA_TAC_Settings.json"
autoSaveSettings: 1
settingsSaveInterval: 30
backupSettings: 1
maxBackupFiles: 5
```

```
  culturalDefaults: 1
  respectCulturalValues: 1
  communityFriendlyOptions: 1
--- !u!114 &1234567894
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 7c6b5d4e3f2a1a0a9e8f7e6d5c4b3a2, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  audioMixer: {fileID: 24100000, guid: 9f8e7d6c5b4a3928190a8b7c6d5e4f3, type:
2}
  masterVolumeParameter: "MasterVolume"
  musicVolumeParameter: "MusicVolume"
  sfxVolumeParameter: "SFXVolume"
  ambientVolumeParameter: "AmbientVolume"
  traditionalMusicParameter: "TraditionalMusicVolume"
  prayerCallParameter: "PrayerCallVolume"
  culturalModeParameter: "CulturalModeVolume"
  prayerTimeParameter: "PrayerTimeMultiplier"
  snapshotNormal: {fileID: 24500006, guid: 9f8e7d6c5b4a3928190a8b7c6d5e4f3,
type: 2}
  snapshotPrayerTime: {fileID: 24500004, guid:
9f8e7d6c5b4a3928190a8b7c6d5e4f3, type: 2}
```

```
  snapshotCulturalEvent: {fileID: 24500008, guid:
9f8e7d6c5b4a3928190a8b7c6d5e4f3, type: 2}
  snapshotCommunityMode: {fileID: 24500010, guid:
9f8e7d6c5b4a3928190a8b7c6d5e4f3, type: 2}
  snapshotRespectfulMode: {fileID: 24500012, guid:
9f8e7d6c5b4a3928190a8b7c6d5e4f3, type: 2}
  currentSnapshot: {fileID: 0}
  respectPrayerTimes: 1
  culturalAudioMode: 1
  communityFriendlyAudio: 1
--- !u!114 &1234567895
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 1234567890}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 6b5c4d3e2f1a0a9a8e7f6e5d4c3b2a1, type:
3}
  m_Name:
  m_EditorClassIdentifier:
  supportedLanguages:
  - "English"
  - "Dhivehi"
  - "Arabic"
  - "Hindi"
  - "Urdu"
  - "Tamil"
  - "Sinhala"
```

- "French"
- "German"
- "Spanish"
- "Chinese"
- "Japanese"
culturalTermsDatabase: {fileID: 11400000, guid:
a8b7c6d5e4f3a2b1a0a9e8f7e6d5c4, type: 2}
traditionalGreetingsDatabase: {fileID: 11400000, guid:
b7c6d5e4f3a2b1a0a9e8f7e6d5c4b3, type: 2}
islamicCalendarDatabase: {fileID: 11400000, guid:
c6d5e4f3a2b1a0a9e8f7e6d5c4b3a2, type: 2}
maldivianCulturalDatabase: {fileID: 11400000, guid:
d5e4f3a2b1a0a9e8f7e6d5c4b3a2a1, type: 2}
currentLanguage: 0
currentCulturalMode: 1
respectCulturalContext: 1
preserveTraditionalMeanings: 1
communityFriendlyTranslations: 1
islamicValueAwareness: 1
localCulturalSensitivity: 1