

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)
    }
}
```



```

        PrayerType Sunrise => "الفجر",
        PrayerType Dhuhr => "الظهر",
        PrayerType Asr => "العصر",
        PrayerType Maghrib => "المغرب",
        PrayerType Isha => "العشاء",
        _ => "الغروب"
    };
}

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) *
monsoonIntensity;
                precipitation[index] = CalculatePrecipitation(pos,
MonsoonType.Northeast);
            }
            // Southwest monsoon (May-October)
            else
            {
                windDirection[index] = new float3(0.8f, 0, -0.6f) *

```

```

monsoonIntensity;
    precipitationIndex] = CalculatePrecipitation(pos,
MonsoonType. 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 :
0.3f;
    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,
Allocator.Persistent);
    var precipitation = new NativeArray<float>(weatherPoints,
Allocator.Persistent);
    var windDirections = new NativeArray<float3>(weatherPoints,
Allocator.Persistent);

    // Generate weather grid
    for (int i = 0; i < weatherPoints; i++)
    {
        float lat = 7f + (i % 32) * 0.1f;
        float lon = -65f + (i % 32) * 0.1f;
        float alt = 100f + (i % 32) * 100f;
        positions[i] = new float3(lat, lon, alt);
        precipitation[i] = Random.Range(0.1f, 1.0f);
        windDirections[i] = RandomUnitVector();
    }
}

```

```

        float Ing = 72f + (i / 32) * 0.1f;
        positions[s] = new float3(lat, 0, Ing);
    }

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

    JobHandle handle = job.Schedule(weatherPoints, 64);
    handle.Complete();
    positionsDispose();
    precipitationDispose();
    windDirectionsDispose();
}
}

```

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++)
        {
            rhythmPatter[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(beatTime * 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,
rhythmHandle);
    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.Exp(-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,
sampleRate, 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,
month) || 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,
conversionHandle);
    eventHandle. Complete();

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

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

public string GetIslamicMonthName(int month)
{
    return month switch
    {
        1 => "Muharram",
        2 => "Safar",
        3 => "Rabī' al-awwal",
        4 => "Rabī' ath-thānī",
        5 => "Jumādá al-ūlá",
        6 => "Jumādá al-ākhirah",
        7 => "Rajab",
        8 => "Sha'bān",
        9 => "Ramadān",
        10 => "Shawwāl",
        11 => "Dhū al-Qa'dah",
    }
}

```

```

        12 => "Dhū al-Hijjah",
        _ => ""
    };
}
}

```

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,
tide);
            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 (سُجُون)
    else if (isEvening)
        return 2; // Yellowfin tuna (سُرِّيْن)
    else
        return 3; // Reef fish (رَبْرَبَة)
}
}

[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,
            Allocator.TempJob);
        var fishTypes = new NativeArray<int>(fishingLocations,
            Allocator.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
            simulation
        }
    }

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

```

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);

        tideLevel[$index] = tide;
        currentFlow[$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); //
        Principal lunar
        float S2 = 0.3f * math.sin(2 * math.PI * t / 12.0f); //
        Principal solar
        float N2 = 0.2f * math.sin(2 * math.PI * t / 12.66f); //
        Larger 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(
angle) * 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) *

```

```

0.3f;
    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.
TempJob);
    var tides = new NativeArray<float>(oceanGrid, Allocator.TempJob
);
    var currents = new NativeArray<float3>(oceanGrid, Allocator.
TempJob);
    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,
tideHandle);
    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);
    islandHeightIndex] = 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)
    float[3] realAtolls = new float[3][]
    {
        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.
TempJob);
        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[
island % atolls]);
}

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

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

```

```

islandtransform.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].
GetComponent<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
floraTypeIndex] = DetermineFloraType(point, height);
floraDensityIndex] = 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 (ންން)
        else
            return 2; // Sea lettuce (ންންންން)
    }
    else if (height < 2.0f) // Coastal area
    {
        if (rainfall > 0.6f)
            return 3; // Breadfruit tree (ންންން)
        else
            return 4; // Tropical almond (ންންންން)
    }
    else // Inland
    {
        return 5; // Banyan tree (ންންންން)
    }
}

[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 *
heightFactor;
}
}

```

```

[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
hours
    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,           // جوز
    SeaLettuce = 2,            // سلاد بحري
    BreadfruitTree= 3,          // نخل
    TropicalAlmond = 4,         // زيزان
    BanyanTree = 5,             // بانيان
    Pandanus = 6,               // پانداني
    SeaHibiscus = 7,            //hibiscus
    BeachMorningGlory = 8,       //hibiscus
    Ironwood = 9,                // خشب آهن
    Mangrove = 10,              // مانغروف
    Seagrass = 11,               // سرگass
    Coral = 12,                  // قoral
}

```

```

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

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

```



```

string GetTraditionalUse(MaldivianFlora flora)
{
    return flora switch
    {
        MaldivianFloraCoconutPalm => "Food, oil, construction
material",
        MaldivianFloraBreadfruitTree => "Staple food, traditional
medicine",
        MaldivianFloraPandanus => "Mat weaving, traditional
crafts",
        MaldivianFloraIronwood => "Boat building, construction",
        MaldivianFloraMangrove => "Coastal protection, fishing
habitat",
        _ => "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;
        newPosition[ 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(
            speed, maxSpeed);

        // Apply steering
        float3 steeringForce= new float3( math.sin(steering), 0,
            math.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[ ] = CalculateTrafficDensity();
    }
}

[ BurstCompile]
float CalculateTrafficDensity()
{

```

```

        // Maldivian traffic: low in morning, peak in evening
        float morningRush = math.saturate( math.sin(( timeOfDay - 7) * PI / 6)) * 0.8f;
        float eveningRush = math.saturate( math.sin(( timeOfDay - 17) * 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,
Allocator.TempJob);
    var velocities = new NativeArray<float3>(activeVehicles,
Allocator.TempJob);
    var steering = new NativeArray<float>(activeVehicles, Allocator
.TempJob);
    var newPositions = new NativeArray<float3>(activeVehicles,
Allocator.TempJob);
    var newVelocities = new NativeArray<float3>(activeVehicles,
Allocator.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.
TempJob),
        timeOfDay = Time.time % 24,
        dayOfWeek = (int)(Time.time / 24) % 7
    };

    JobHandle physicsHandle = physicsJob.Schedule(activeVehicles,
16);
    JobHandle trafficHandle= trafficJob.Schedule(physicsHandle);
}

```


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,
timeOfDay);
    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=shopping, 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.
Length); 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.TempJob);
    var needs = new NativeArray<float>(population, Allocator.TempJob);
    var schedules = new NativeArray<float>(population, Allocator.TempJob);
    var destinations = new NativeArray<float3>(population,
Allocator.TempJob);
    var activities = new NativeArray<float>(population, Allocator.TempJob);

    // Initialize population data
    for (int i = 0; i < population; i++)
}

```

```

    {
        character[i] = GenerateRandomCharacter(i);
        positions[i] = GetRandomIslandPosition();
        needs[i] = UnityEngine.Random.Range(0f, 1f);
        schedule[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.TempJob),
        population= population,
        islandSize= 100f
    };

    JobHandle behaviorHandle = behaviorJob.Schedule(population, 16);
    JobHandle populationHandle = populationJob.Schedule(
behaviorHandle);
    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),
    };
}

```



```

        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
            strength
            territoryControl[index] = CalculateTerritoryInfluence(
                center, 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 -
distance * 0.1f);

        influence = proximityInfluence * (strength -
otherStrength);
    }

    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 -
policeReduction);
}

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,
Allocator.TempJob);
    var gangStrengths = new NativeArray<float>(gangCount, Allocator
.TempJob);
    var territoryControl = new NativeArray<float>(gangCount,
Allocator.TempJob);
    var activityLevels = new NativeArray<float>(gangCount,
Allocator.TempJob);
}

```

```

// Initialize gangs
for (int i = 0; i < gangCount; i++)
{
    gangs[i] = GenerateMaldivianGang(i);
    territoryCenters = 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
);
activityHandleComplete();

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

territoryCentersDispose();
gangStrengths Dispose();
territoryControlDispose();
activityLevelsDispose();
}

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

```



```

        "Thulusdhoo", "Himmafushi", "Huraa", "Maafushi", "Gulhi",
        "Guraidhoo", "Fulidhoo", "Keyodhoo", "Rakeedhoo",
    "Thinadhoo",
        "Kudahuvadhoo", "Veymandoo", "Maaenboodhoo", "Bileydhoo",
    "Gadhdhoo",
        "Fonadhoo", "Dhanbidhoo", "Maamendhoo", "Kunahandhoo",
    "Dhiyamingili",
        "Madaveli", "Hoandeddhoo", "Rathafandhoo", "Vaadhoo",
    "Kolamaafushi",
        "Kanduhulhudhoo", "Nilandhoo", "Dhadimago", "Biledhdhoo",
    "Dhoondigan",
        "Faresmaathodaa", "Magoodhoo", "Dhevvadhoo",
    "Kondeymatheela", "Dhiyaree",
        "Fyoaree", "Maamendhoo", "Dhevva", "Fyoaree", "Kanduvale"
    ,
        "Bodufolhudhoo", "Feridhoo", "Mathiveri", "Bathalaa",
    "Kudafolhudhoo",
        "Maalhos", "Rasdho", "Ukulhas", "Mathiveri", "Feridhoo",
        "Bodufolhudhoo", "Himandhoo", "Thoddoo", "Feridhoo",
    "Maalhos",
        "Rasdho", "Ukulhas", "Mathiveri", "Bathalaa",
    "Kudafolhudhoo",
        "Maamigili", "Dhigurah", "Fenfushi", "Mahibadhoo",
    "Dhangethi"
    };
}
}

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 protection", "Intimidation", "Petty theft" },
        MaldivianGang.GangType.DrugCartel => new string[] { "Drug trafficking", "Money laundering", "Corruption" },
        MaldivianGang.GangType.SmugglingRing => new string[] { "Goods smuggling", "Human trafficking", "Tax evasion" },
        MaldivianGang.GangType.ProtectionRacket => new string[] { "Extortion", "Security services", "Threats" },
        _ => new string[] { "Illegal activities", "Territory disputes", "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[$] = traditionalColors[i % traditionalColors.Length];
        buildingHeights[$] = 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,
Allocator.TempJob);
    var terrainHeights = new NativeArray<float>(buildingCount,
Allocator.TempJob);
    var buildableAreas = new NativeArray<bool>(buildingCount,
Allocator.TempJob);
    var buildingTypes = new NativeArray<int>(buildingCount,
Allocator.TempJob);
    var buildingColors = new NativeArray<float3>(buildingCount,
Allocator.TempJob);
    var buildingHeights = new NativeArray<float>(buildingCount,
Allocator.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],
buildingTypes[i], buildingColors[i], buildingHeights[i]);
    }
}

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

MaldivianBuilding CreateBuilding(int index, float3 position, int
type, float3 color, float height)
{
    MaldivianBuilding BuildingType buildingType = (
    MaldivianBuilding.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),
    };
}

```



```

        "Religious", "Community", "Education" },
        MaldivianBuildingBuildingType.School => new string[] {
    "Education", "Learning", "Childcare" },
        MaldivianBuildingBuildingType.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;
            nextMissionID[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++)
        {
            reward[i] = CalculateReward(missionTypes[i],
difficultyLevels[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,
Expert = 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,
Allocator.TempJob);
var completionProgress = new NativeArray<float>(missionCount,
Allocator.TempJob);
var missionComplete = new NativeArray<bool>(missionCount,
Allocator.TempJob);
var nextMissionIDs = new NativeArray<int>(missionCount,
Allocator.TempJob);
var rewards = new NativeArray<float>(missionCount, Allocator.
TempJob);
var missionTypes = new NativeArray<int>(missionCount, Allocator
.TempJob);
var difficultyLevels = new NativeArray<int>(missionCount,
Allocator.TempJob);

// Initialize missions
for (int i = 0; i < missionCount; i++)
{
    mission[i] = GenerateMaldivianMission(i);
    missionState[i] = (int)missions[i].type;
    completionProgress[i] = 0f;
    missionType[i] = (int)missions[i].type;
    difficultyLevel[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(
missionCount, 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.MissionType[])
System.Enum.GetValues(typeof(MaldivianMission.MissionType));
        MaldivianMission.MissionType randomType = types[index % types.Length];

        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
    {
        MaldivianMissionMissionType. Story => new string[] {
            "Investigate the mystery", "Gather clues", "Confront the truth",
            MaldivianMissionMissionType. Delivery => new string[] {
                "Pick up package", "Deliver to destination", "Avoid detection",
                MaldivianMissionMissionType. Assassination => new string[] {
                    "Locate target", "Plan approach", "Eliminate target",
                    MaldivianMissionMissionType. Collection => new string[] {
                        "Find items", "Collect resources", "Return to client",
                        _ => new string[] { "Complete main objective", "Avoid complications", "Report back" }
                    };
                };
            };
        };

        Vector3 GetRandomLocation()
        {
            return new Vector3( UnityEngine. Random. Range( 2f , 7f ), 0,
                UnityEngine. 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
            {
                MaldivianMissionMissionType. Emergency => 300f , // 5
                minutes
                MaldivianMissionMissionType. Racing => 600f , // 10
                minutes
                MaldivianMissionMissionType. Delivery => 1800f , // 30
                minutes
                _ => 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(
state, 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.
Length; 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,
Allocator.TempJob);
    var playerChoices = new NativeArray<int>(dialogueCount,
Allocator.TempJob);
    var nextNodes = new NativeArray<int>(dialogueCount, Allocator.
TempJob);
    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",
"Elder", "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" => "ଓମ୍ବାର୍ଜନ୍ଦି",
            "Fisherman" => "ଫିଶରମାନ୍",
            "Shopkeeper" => "ଶୋପକିନ୍ଡିର୍",
            "Elder" => "ଅଧିକାରୀ",
            "Youth" => "ଯୁବତୀ",
            "Official" => "ଅଧିକାରୀ କମିଶନ୍ରେଟର୍",
            "Tourist" => "ଟୂରିସ୍ଟ୍",
            "ReligiousLeader" => "ଧ୍ୟାନିକ ନିର୍ଦ୍ଦେଶୀର୍ଣ୍ଣାତ୍ମୀୟ",
            _ => "ମହାନ୍"
        };
    }

    string GenerateDialogueText( int index, string speaker)
    {
        string[] templates = {
            "Welcome to our island, traveler.",
            "The sea has been rough lately."
        }
    }
}

```



```

        effects new string[ 0 ],
        relationshipImpact = 0,
        endsDialogue = true
    }
};

}

string GetRandomEmotion()
{
    string[] emotions = { "neutral", "happy", "sad", "angry",
"surprised", "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,
index );
        }
    }
}

```

```

    }

    [ BurstCompile]
    int CalculateOptimalSlot(float size, float weight, int
itemIndex)
{
    // 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
value
            bool isTraditional = i % 3 == 0;
            isTraditionalItems[i] = isTraditional;
            itemValue$[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
inventory

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.
TempJob);
var itemWeights = new NativeArray<float>(totalItems, Allocator.
TempJob);
var optimalSlots = new NativeArray<int>(totalItems, Allocator.
TempJob);
var itemValues = new NativeArray<float>(totalItems, Allocator.
TempJob);
var itemCategories = new NativeArray<int>(totalItems, Allocator.
.TempJob);
var isTraditionalItems = new NativeArray<bool>(totalItems,
Allocator.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(
totalItems, 32);
JobHandle generationHandle = generationJob.Schedule(
optimizationHandle);

```

```

        generationHandle Complete();

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

    itemSizes Dispose();
    itemWeights Dispose();
    optimalSlots Dispose();
    itemValues Dispose();
    itemCategories Dispose();
    isTraditionalItemsDispose();
}

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

    return new InventoryItem
    {
        itemID = index,
        itemName = GenerateItemName(index, randomCategory),
        dhivehiName = GenerateDhivehItemName(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 category)
{
    return category switch

```



```

religious 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[] {
            "Efficiency: 80%", "Durability: 150"
        },
        InventoryItem.ItemCategory.Food => new string[] {
            "Nutrition: 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,
supply, 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 +
fluctuation );
    }
}

[BurstCompile]

```

```

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

    public void Execute()
    {
        GenerateMaldivianTradeNetwork();
    }

    [BurstCompile]
    void GenerateMaldivianTradeNetwork()
    {
        // Real Maldivian trade centers
        float[3] 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.Length; i++)
        {
            tradeCenters[i] = centers[i];
            routeValues[i] = 1000.0f + i * 100.0f; // Trade value decreases 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.TempJob);
    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.TempJob);
    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++)
    {
        marketItem[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",
    "Textiles", "Electronics", "Fuel", "Medicine", "ConstructionMaterials",
    "TourismServices" };
    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,
    };
}

```


}

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.
CreateFromIndex((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
            steady
        }
        else
        {
            weaponDamages[i] = 20.0f + i * 3.0f; // Modern
            weapons: 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
weapons

    var attackPowers = new NativeArray<float>(weaponCount,
Allocator. TempJob);
    var defenseValues = new NativeArray<float>(weaponCount,
Allocator. TempJob);
    var accuracyValues = new NativeArray<float>(weaponCount,
Allocator. TempJob);
    var damageValues = new NativeArray<float>(weaponCount,
Allocator. TempJob);
    var hitSuccess = new NativeArray<bool>(weaponCount, Allocator.
TempJob);
    var weaponDamages = new NativeArray<float>(weaponCount,
Allocator. TempJob);
    var weaponSpeeds = new NativeArray<float>(weaponCount,
Allocator. TempJob);
    var isTraditionalWeapons = new NativeArray<bool>(weaponCount,
Allocator. 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(
damageHandle);
    traditionalHandleComplete();

// 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
}

```



```

        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",
"Knockback" },
            WeaponType.MolotovCocktail => new string[] { "Fire damage",
"Area denial" },
            WeaponType.SmokeBomb => new string[] { "Vision obscurement",
"Stealth" },
            WeaponType.StunGun => new string[] { "Stun effect", "Non-
lethal" },
            _ => 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,
enemyPos, 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,
float visibility)
{
    float distance = math.length( player - enemy );
    float maxRange = 15.0f;
    float normalizedDistance = math.saturate( distance /
maxRange );

    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);
        concealmentValue[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 +
spot.z * 0.1f) * 0.3f + 0.5f;

    return math.saturate(heightConcealment +
environmentConcealment);
}
}

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,
Allocator.TempJob);
    var enemyPositions = new NativeArray<float3>(stealthChecks,
Allocator.TempJob);
    var visibilityLevels = new NativeArray<float>(stealthChecks,
Allocator.TempJob);
    var detectionStatus = new NativeArray<bool>(stealthChecks,
Allocator.TempJob);
    var detectionMeters = new NativeArray<float>(stealthChecks,
Allocator.TempJob);
    var hidingSpots = new NativeArray<float3>(20, Allocator.TempJob);
    var concealmentValues = new NativeArray<float>(20, Allocator.TempJob);

    // Initialize positions
    for (int i = 0; i < stealthChecks; i++)
    {
        playerPositions[i] = new float3(UnityEngine.Random.Range(-
10f, 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), 0, UnityEngine.Random.Range(-5f, 5f));
    }
}

```

```

        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
};

, 16);
JobHandle detectionHandle = detectionJob.Schedule(stealthChecks
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" => "ଓଡ଼ିଆକୁଣ୍ଡର",
        "Fisherman" => "ଓଡ଼ିଆକୁଣ୍ଡର",
        _ => "ଓଡ଼ିଆ"
    };
}
}

```

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++)
        {
            wantedLevel[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 minute 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,
Allocator.TempJob);
    var patrolRadii = new NativeArray<float>(unitCount, Allocator.
TempJob);
    var patrolDestinations = new NativeArray<float3>(unitCount,
Allocator.TempJob);
    var responseTimes = new NativeArray<float>(unitCount, Allocator
. TempJob);
    var crimeSeverities = new NativeArray<int>(10, Allocator.
TempJob);
    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;
}

PoliceUnit GeneratePoliceUnit(int index)
{
    string[] stations = { "Malé HQ", "Addu Station", "Hithadhoo Outpost", "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),
    }
}

```

```

        "Naifarū 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 ] ^ 0xAA);
        }
    }

    [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 = "ଡିବେହିପାଇନ୍"
    };
}

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

int saveDataSize = 1000;
var uncompressedData = new NativeArray<byte>( saveDataSize,
Allocator. TempJob);
var compressedData = new NativeArray<byte>( saveDataSize,
Allocator. TempJob);
var localData = new NativeArray<byte>( saveDataSize, Allocator.
TempJob);
var cloudData = new NativeArray<byte>( saveDataSize, Allocator.
TempJob);
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(
saveDataSize, 64);
    JobHandle validationHandle = validationJob. Schedule(
compressionHandle);
    validationHandle Complete();

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

StartAutoSave();
}

void StartAutoSave()
{
    if ( autoSaveEnabled)
    {
        InvokeRepeating( "AutoSave",  autoSaveInterval,
autoSaveInterval);
    }
}

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(monitoredPoints, 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
#ifndef 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' => 'fiä',
        'E' or 'e' => 'fiá',
        'F' or 'f' => 'fiÑ',
        'G' or 'g' => 'fiÇ',
        'H' or 'h' => '?',
        'I' or 'i' => 'fià',
        'J' or 'j' => 'fiää',
        'K' or 'k' => 'fiâ',
    }
}

```

```
'L' or 'l' => 'fiÚ',  
'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 = "fiàfiްfiÉfi™fiéfiްfiāfiް fiàfiްfiÉfi™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.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
    Initialize AudioSource Pools();

    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 wildlifelIntensity = new NativeArray<float>(5, Allocator.TempJob);
    var culturallIntensity = 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 Initialize AudioSource Pools()
{
    // Create pools for different audio types
    ambientSources = new AudioSource[8];
    culturalSources = new AudioSource[6];
    sfxSources = new AudioSource[16];

    Create AudioSource Pool("AmbientPool", ambientSources, ambientGroup);
    Create AudioSource Pool("CulturalPool", culturalSources, culturalGroup);
    Create AudioSource Pool("SFXPool", sfxSources, sfxGroup);
}

void Create AudioSource Pool(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();
    }
}

 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 (
```

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 isValid = validationResults[0];
Debug.Log($"Cultural data validation: {isValid}");

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 = JsonUtilityToJson(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,
    };
}
```

```
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)  
        {  
            // Implement validation logic here  
        }  
    }  
}
```

```
        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;
}

[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 actionPerformed = actionTypes[index];
            float context = culturalContext[index];

            // Calculate reputation change based on cultural appropriateness
        }
    }

    [BurstCompile]
    struct FinalizeJob : IJob
    {
        [ReadOnly] public NativeArray<float> finalReputation;
        [ReadOnly] public NativeArray<float> reputationChanges;
        [ReadOnly] public NativeArray<float> culturalMultipliers;
        [ReadOnly] public NativeArray<float> currentReputations;
        [ReadOnly] public NativeArray<int> actionTypes;
        [ReadOnly] public NativeArray<float> culturalContexts;

        public void Execute()
        {
            for (int i = 0; i < finalReputation.Length; i++)
            {
                finalReputation[i] = currentReputation[i] + reputationChanges[i];
                finalReputation[i] *= culturalMultipliers[i];
            }
        }
    }
}
```

```
    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 (GetAveragel IslandReputation(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 GetAveragel IslandReputation(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 = skillLvl,
learningRates = learningRates,
newSkillLevels = newSkillLvl,
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();
skillLvl.Dispose();
learningRates.Dispose();
newSkillLvl.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 = $"Skill achieved! {skillName} - Level {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;
}
```