LAST LIGHT — DEMO TECHNICAL DESIGN DOCUMENT (TDD)

Scope: Demo vertical slice only
Engine: Unity (2D URP)
Version: TDD Demo v1.0

---

1. TECHNICAL OVERVIEW

1.1 Demo Scope (Technical)

The demo requires:

- One main gameplay scene with:
    - Player controller (top-down 2D)
    - Single enemy type (zombie walker)
    - Resource nodes (wood, stone, metal)
    - Base structure
    - One turret type
    - Wave system
    - Shared ammo system
- Simple UI HUD & menus
- Optional simple save/load (can be stubbed if needed)

No multi-scene streaming, no netcode, no deep meta-progression.

---

2. PROJECT STRUCTURE

2.1 Folder Layout

Recommended Unity project structure:

- `Assets/`
    - `Art/`
        - `Characters/Player`
        - `Characters/Zombie`
        - `Environment/Tiles`
        - `Environment/Props`
        - `VFX`
        - `UI`
    - `Audio/`
        - `SFX`
        - `Music`
    - `Prefabs/`
        - `Characters`
        - `Environment`
        - `Turrets`
        - `Resources`

        - UI
        - Managers
    - Scenes/
        - Boot
        - MainMenu
        - DemoLevel
    - Scripts/
        - Core/ (base classes, services, utilities)
        - Player/
        - Enemies/
        - Combat/
        - Resources/
        - BaseAndTurrets/
        - Waves/
        - UI/
        - Audio/
        - Data/
    - ScriptableObjects/
        - Config/ (balance data, wave data, etc.)

- `Settings/` (URP pipelines, Input System, etc.)

2.2 Naming Conventions

- C# files: `PascalCase`, `PlayerController.cs`, `ZombieAIController.cs`
- GameObjects: `Player`, `Zombie_Walker`, `BaseCore`, etc.
- ScriptableObjects: `WaveConfig_Wave1`, `PlayerStatsConfig`, etc.

---

3. CORE ARCHITECTURE & PATTERNS

3.1 Main Systems

- **Player System**
- **Enemy / AI System**
- **Wave System**
- **Combat & Damage System**
- **Resource System**
- **Base & Turret System**
- **Shared Ammo System**
- **UI System**
- **Audio System**
- **(Optional) Save/Load System**

3.2 Key Interfaces & Base Classes

Define reusable interfaces:

```
public interface IDamageable
{
    void TakeDamage(float amount);
    bool IsAlive { get; }
}

public interface IResourceNode
{
    void Harvest(int amount); // or void Harvest();
    ResourceType ResourceType { get; }
    int RemainingAmount { get; }
}

public interface IInteractable
{
    void Interact(PlayerController player);
    bool CanInteract(PlayerController player);
}
```

Base classes:

- `HealthComponent`
- `CharacterBase` (shared movement/health hooks if needed)
- `ProjectileBase`
- `TurretBase`

3.3 Managers & Singletons

Use **MonoBehaviour-based managers**:

- `GameManager`
- `WaveManager`

- `AmmoManager`
- `UIManager`
- `AudioManager`

Use a simple pattern:

- Static instance (singleton) for each manager
- Managers created via a `Managers` prefab in the scene or loaded in Boot scene

---

4. SCENE & FLOW MANAGEMENT

4.1 Scenes

- `Boot`:
    - Loads `MainMenu` or jumps straight to `DemoLevel` in dev builds.
- `MainMenu`:
    - Simple menu to start demo, open settings, and quit.
- `DemoLevel`:
    - The whole demo game (map, player, enemies, etc.)

4.2 Scene Flow

1. Launch → `Boot` scene
2. Load `MainMenu` → player presses *Start Demo*
3. Load `DemoLevel`

4. When demo ends (base destroyed / final wave cleared):
    - Show end screen UI
    - Option: return to MainMenu

Scene loading can use `SceneManager.LoadScene`, no async/streaming necessary for demo.

---

5. INPUT SYSTEM

Use Unity's new Input System or legacy Input Manager — but pick one.

5.1 Controls (Demo)

- Move: WASD
- Aim: Mouse position
- Fire gun: Left Mouse Button
- Melee: Right Mouse Button
- Interact: `E`
- Reload (optional): `R`
- Pause: `Esc`

5.2 Input Handling

Implement `PlayerInputHandler`:

- Handles input actions and passes them to `PlayerController`:
    - `Vector2 movementInput`
    - `Vector2 aimDirection` from world position of mouse

- `bool firePressed`
- `bool meleePressed`
- `bool interactPressed`

---

6. PLAYER SYSTEM

6.1 PlayerController

Responsibilities:

- Read input (from `PlayerInputHandler`)
- Move the character (rigidbody or character controller)
- Rotate/face aim direction
- Trigger melee and ranged attacks
- Track and modify player HP

Components on Player GameObject:

- `Rigidbody2D` (dynamic; or kinematic if manually moving)
- `Collider2D` (capsule/box)
- `PlayerController.cs`
- `PlayerHealth.cs` (implements `IDamageable`)
- `Animator` + `AnimationController`
- `SpriteRenderer`
- Optional `AudioSource` for player-centric SFX

6.2 Movement

- Physics or move via `Rigidbody2D.MovePosition` in `FixedUpdate`.
- Movement speed from `PlayerStatsConfig` ScriptableObject.

```
public class PlayerController : MonoBehaviour
{
    public float moveSpeed;
    private Rigidbody2D rb;
    private Vector2 moveInput;

    void FixedUpdate()
    {
        rb.MovePosition(rb.position + moveInput * moveSpeed *
Time.fixedDeltaTime);
    }
}
```

6.3 Ranged Attack Implementation

- `GunController` handles:
    - Check `AmmoManager` for available ammo.
    - Instantiate or reuse bullet via pool.
    - Set direction towards aim.
    - Apply damage via bullet collision.

6.4 Melee Attack Implementation

- Use a short-lived **attack collider** or physics overlap (`OverlapCircle`/`OverlapBox`) in front of player during attack window.
- For each hit `IDamageable` found, call `TakeDamage()`.

---

7. COMBAT & DAMAGE

7.1 Damage System

Use `IDamageable` + `HealthComponent`:

```
public class HealthComponent : MonoBehaviour, IDamageable
{
    public float maxHealth;
    private float currentHealth;

    public bool IsAlive => currentHealth > 0;

    public void TakeDamage(float amount)
    {
        currentHealth -= amount;
        if (currentHealth <= 0) Die();
    }

    void Die()
    {
        // Notify, play FX, etc.
    }
}
```

7.2 Bullets / Projectiles

`ProjectileBase.cs`:

- Holds speed, damage, lifetime.
- On collision:
    - Check for `IDamageable`.
    - Call `TakeDamage(damage)`.

    - Despawn (return to pool).

Use **object pooling** for bullets.

---

8. ENEMY / AI SYSTEM

8.1 Zombie Prefab

Components:

- `Rigidbody2D`
- `Collider2D`
- `ZombieAIController.cs`
- `HealthComponent.cs` (implements `IDamageable`)
- `Animator` + `AnimationController`
- `SpriteRenderer`

8.2 AI States

Implement a simple state machine inside `ZombieAIController`:

States:

- `Idle` / `Wander`
- `ChasePlayer`
- `ChaseBase`
- `Attack`

- `Dead`

Logic:

- Check distance to player and base each frame.
- If player within detection radius → chase player.
- Else if base within detection radius → chase base.
- If in attack range → attack state.

8.3 Pathfinding

For demo, two options:

- **Simple direct movement** (zombies move directly towards target using normalized vector)
- OR **Unity's NavMesh for 2D** (adds complexity but avoids obstacle issues)

For first demo, **simple direct movement** is acceptable if level design avoids complex obstacles.

---

9. WAVE SYSTEM

9.1 WaveManager

`WaveManager.cs` responsibilities:

- Store list of `WaveConfig` data (ScriptableObjects).
- For each wave:
    - Spawn zombies according to config.
    - Track active enemies.
    - When wave cleared → trigger intermission.

- Notify UI on wave start/end.

9.2 WaveConfig ScriptableObject

```
[CreateAssetMenu(menuName = "LastLight/WaveConfig")]
public class WaveConfig : ScriptableObject
{
    public int waveNumber;
    public int zombieCount;
    public float spawnInterval;
    public float zombieHpMultiplier;
    public float zombieSpeedMultiplier;
}
```

Each wave spawns `zombieCount` zombies, spaced by `spawnInterval`, and modifies base zombie stats via multipliers.

9.3 Spawners

Use multiple `ZombieSpawner` points in the level:

- Each spawner has radius or points where zombies appear.
- `WaveManager` distributes spawns across spawners.

---

10. RESOURCE SYSTEM

10.1 Resource Types

Define an enum:

```
public enum ResourceType
{
    Wood,
    Stone,
    Metal
}
```

10.2 ResourceNode

`ResourceNode.cs` implements `IResourceNode` & `IInteractable`:

- Fields:
    - `ResourceType resourceType;`
    - `int maxAmount;`
    - `int remainingAmount;`
- On `Interact`:
    - Reduce `remainingAmount`
    - Add to player inventory
    - Play FX
- When `remainingAmount <= 0`:
    - Disable node or destroy, optionally schedule respawn.

10.3 Resource Inventory

`ResourceInventory.cs` on player or global:

- Holds counts:
    - `int wood;`
    - `int stone;`
    - `int metal;`
- Methods:

    - `AddResource(ResourceType type, int amount)`
    - `bool TryConsume(ResourceType type, int amount)`
- Raise events on change for UI updates.

---

11. BASE & TURRET SYSTEM

11.1 BaseCore

`BaseCore.cs`:

- Attached to base GameObject, with `HealthComponent` or custom HP.
- Implements `IDamageable`.
- On `TakeDamage`, triggers base damage FX and UI updates.
- On `Die`, informs `GameManager` to trigger demo over.

11.2 Repair Mechanic

`BaseRepairInteractable.cs` implements `IInteractable`:

- When `Interact(player)`:
    - Checks player `ResourceInventory`.
    - If sufficient resources:
        - Calculate heal amount.
        - Reduce resources.
        - Increase base HP.
- Optionally use a “hold to repair” with coroutine and UI progress bar.

11.3 Turret System

Turret prefab components:

- `TurretController.cs`
- `CircleCollider2D` or radius indicator (gizmos)
- `SpriteRenderer`
- Optional `AudioSource`

Turret Placement:

- Use `TurretPlacementSystem.cs` attached to player or global manager:
    - On build command (`E` near build node), check:
        - Player resources vs turret cost.
        - Placement grid validity.

Turret Behavior:

- In `Update()`:
    - Search for nearest zombie in range (Physics2D.OverlapCircle or track via events).
    - If target found and cooldown ready, request ammo from `AmmoManager`:
        - If ammo available:
            - Fire bullet (from pool).
            - Reset cooldown.

---

12. SHARED AMMO SYSTEM

12.1 AmmoManager

Global manager: `AmmoManager.cs`

- Fields:
    - `int currentAmmo;`
    - `int maxAmmo;`
- Methods:
    - `bool TryConsumeAmmo(int amount)`
    - `void AddAmmo(int amount)`
    - `int GetCurrentAmmo()`

Used by:

- `GunController` (player shooting)
- `TurretController` (turret firing)

12.2 Player & Turret Integration

- Player firing:
    - Calls `AmmoManager.TryConsumeAmmo(1)` before shooting.
- Turret firing:
    - Same call inside turret update when ready to shoot.

12.3 UI Binding

- `AmmoUIWidget` subscribes to ammo changed events:

    - Shows `currentAmmo / maxAmmo`.

---

13. UI ARCHITECTURE

13.1 UIManager

`UIManager.cs`:

- Central point for:
    - HUD references (health, ammo, resources, wave text).
    - Showing/hiding panels (pause, end screen).
    - Receiving events from gameplay to update labels.

13.2 HUD Structure

- `Canvas_HUD`:
    - `HealthBar`
    - `AmmoCounter`
    - `ResourcePanel` (Wood/Stone/Metal)
    - `WaveIndicator`
    - `InteractPrompt`

13.3 UI Updating Pattern

- Gameplay systems fire events or call `UIManager` methods:
    - `OnPlayerHealthChanged(float current, float max)`

    - `OnAmmoChanged(int current, int max)`
    - `OnResourcesChanged(ResourceType type, int amount)`
    - `OnWaveChanged(int waveNumber)`

UI subscribes to these events via `UnityEvent` or C# events.

---

14. AUDIO SYSTEM

14.1 AudioManager

`AudioManager.cs`:

- Holds references to:
    - SFX libraries (ScriptableObjects or serialized arrays)
    - Music tracks and mixers
- Methods:
    - `PlaySFX(SFXType type, Vector3 position)`
    - `PlayMusic(MusicTrack track)`
    - `StopMusic()`

14.2 Audio Hooks

- Player:
    - Footsteps, gunshots, melee hits, hurt.
- Zombies:
    - Idle groans, attack, hit, death.

- Environment:
    - Harvest hits, turret fire, base damage.
- UI:
    - Button clicks, hover, confirm.

Gameplay scripts call `AudioManager` instead of adding logic directly in audio code.

---

15. SAVE/LOAD (DEMO-OPTIONAL)

For the demo, this can be **minimal or skipped**. If included:

15.1 SaveData Model

```
[System.Serializable]
public class DemoSaveData
{
    public int currentWave;
    public int playerHealth;
    public int baseHealth;
    public int ammoCount;
    public int wood;
    public int stone;
    public int metal;
    // (Optional) turret placements
}
```

15.2 Save System

`SaveSystem.cs`:

- Methods:
    - `void SaveGame(DemoSaveData data)`

    - `DemoSaveData LoadGame()`

- Implementation:
    - JSON → write to `Application.persistentDataPath`.

---

16. PERFORMANCE & OPTIMIZATION

16.1 Pooling

Pool:

- Bullets
- Zombies (if needed)
- Common VFX (hit, death)

Use a simple `ObjectPool` class with:

- `Get()`
- `Return()`

16.2 Physics

- Limit `FixedUpdate` complexity — no heavy per-frame pathfinding.
- Use simple vector movement.

16.3 Rendering

- Use sprite atlases.
- Limit overdraw (avoid huge transparent sprites everywhere).

---

17. DEBUGGING & DEV TOOLS

17.1 Dev Cheats (Editor-Only)

- Spawn ammo
- Spawn resources
- Skip to next wave
- Toggle god mode

Implement via:

- Simple debug UI in development builds
- Keyboard shortcuts (e.g., `F1`, `F2`, etc.)

17.2 Logging

Use conditional logging:

```
#if UNITY_EDITOR
Debug.Log("Wave started: " + waveNumber);
#endif
```

Or a custom `Logger` with verbosity levels.

---

18. INTEGRATION ORDER (BUILD ORDER)

Recommended implementation order for demo:

1. **Core architecture & managers** (`GameManager`, `UIManager`, `AmmoManager`, `WaveManager` stub)

2. **Player movement + camera**
3. **Ranged combat (gun + bullets)**
4. **Zombie AI + basic waves**
5. **Damage system & health**
6. **Resource nodes + inventory**
7. **Base core + repair**
8. **Turrets (placement + firing)**
9. **Shared ammo integration** (player + turrets)
10. **UI hook-up** (HUD, wave text, prompts)
11. **Audio pass**
12. **Polish & optimization**