Survivor's Journey: A VR Decision-Making Survival Game

Md AL Samiul Amin Rishat
Department of Information Science Concentration in Data Science
University of North Texas, Texas, Denton
mdalsamiulaminrishat@my.unt.edu

ABSTRACT

This paper presents the design, architecture, and implementation of *Survivor's Journey*, a 3D single-player survival game developed in Unity 6 using the Universal Render Pipeline (URP). The game places the player as the sole survivor on an uncharted island, tasked with gathering resources, crafting tools, and building a raft to escape. The paper details the core systems developed, including survival mechanics, inventory management, a crafting pipeline, dynamic day/night cycling, resource interaction with animation, and a layered UI architecture. Key engineering challenges such as Unity Input System integration, event-driven UI updates, ScriptableObject-based data design, and game state management are discussed. The result is a complete, playable prototype demonstrating interconnected game systems built with modular, extensible C# architecture.

Keywords: Unity, game development, survival game, C#, URP, ScriptableObject, crafting system, resource interaction

I. INTRODUCTION

Survival games represent one of the most engaging genres in interactive entertainment, requiring players to manage resources, adapt to dynamic environments, and pursue a clear long-term objective. This project, titled Survivor's Journey, was conceived as a full-stack game development exercise using Unity 6 LTS, targeting a third-person survival experience set on an island environment.

The player begins with no items and must explore the island to gather wood, stone, and vine, craft a stone axe using the in-game crafting system, collect further resources, and ultimately construct a raft to escape. The game ends in one of two outcomes: escape (victory) or death from hunger and exhaustion (game over).

The primary contribution of this paper is a documented account of the technical design decisions made throughout development, including the challenges of integrating Unity's new Input System with third-party character controllers, implementing event-driven UI systems, and managing complex game state through a lightweight manager architecture.

The remainder of this paper is structured as follows. Section II reviews related work. Section III describes the system architecture. Section IV covers gameplay mechanics. Section V presents the UI and interaction systems. Section VI discusses results. Section VII concludes with future work.

II. RELATED WORK

Recent studies on survival and open-world games show that this genre depends on resource use, exploration, environmental interaction, and player decision-making. Strannegård et al. [1] described survival games as games where players must search for resources while avoiding obstacles or threats. May [2] proposed the concept of "emergent ecological dynamics," explaining that meaning in games can emerge from player interaction with simulated environments. These ideas are relevant to Survivor's Journey because the island is not only a setting, but also the main gameplay system where the player gathers resources, survives, crafts tools, and works toward escape.

Some research also highlights the importance of open-world freedom and player autonomy. Anto et al. [3] found that open-world games provide immersive environments, nonlinear gameplay, and freedom of exploration, which can support player engagement and relaxation. This supports the design of Survivor's Journey, where the player explores the island at their own pace, collects materials, and makes decisions about crafting and progression.

Crafting and progression systems have also been examined in recent serious-game research. Filippas and Xinogalos [4] designed and evaluated Elementium, a serious game that used game mechanics such as quests, progress systems, and interactive tasks to support learning.

Their study shows how structured game systems can guide player progress and maintain engagement. Similarly, Survivor’s Journey uses crafting recipes, inventory slots, and raft components to guide the player from basic resource collection toward the final escape objective.

In terms of development tools, Moiseienko et al. [5] showed that Unity is suitable for game-development education because it provides a low barrier to entry, rich features, cross-platform support, and wide industry adoption. Gazis and Katsiri [6] also reviewed game engines and discussed Unity as one of the most widely used engines for digital game development. These studies support the use of Unity 6 in Survivor’s Journey, especially because the project requires 3D rendering, animation, player input, UI, inventory, crafting, and game-state management within one engine.

Recent Unity-related studies also support modular and reusable development practices. Chebanyuk [7] proposed an approach for reusing Unity game scenes, showing the importance of reuse in Unity-based projects. Jackson and Rosero [8] proposed a five-stage Unity development framework that includes design considerations and data management, showing how structured development can reduce complexity. These ideas relate to Survivor’s Journey, where resource interaction, inventory management, crafting, UI updates, day/night cycling, and game-state control are separated into modular C# systems.

Previous studies show that survival games benefit from environmental interaction, resource-based progression, open-world exploration, crafting systems, and modular software architecture. Survivor’s Journey applies these concepts by combining survival mechanics, inventory management, crafting, resource interaction, UI systems, and game-state management into a complete Unity-based prototype.

III. SYSTEM ARCHITECTURE

The game begins with a simple main menu that introduces the title and provides a direct entry point into gameplay, as shown in Figure 1. This menu establishes the visual identity of *Survivor’s Journey* before the player enters the island environment.

Fig. 1. Main menu of Survivor's Journey showing game title and play button.

A. Technology Stack

The game was built in Unity 6 LTS using the Universal Render Pipeline for improved lighting fidelity. The scripting language is C# with the .NET Standard 2.1 profile. The Unity Input System package (version 1.7) was used for all player input. TextMeshPro was used for all UI text rendering. The Unity Starter Assets third-person character controller provided the base locomotion system.

B. Project Structure

Scripts are organised into five responsibility areas: Core (player stats, interaction, Starter Assets integration), Inventory (item data, inventory management), Crafting (recipe data, crafting manager), World (resource nodes, campfire, day/night cycle, escape trigger, raft spawner), and UI (HUD, inventory HUD, crafting UI, notifications, game over, ending panel, pause menu).

C. ScriptableObject Data Architecture

All game items are defined as ItemData ScriptableObjects, and all crafting recipes are defined as CraftingRecipe ScriptableObjects. This design decouples data from logic: the InventorySystem operates on ItemData references without knowing concrete item types, the CraftingManager validates recipes against the inventory generically, and the InventoryHUD subscribes to an onInventoryChanged UnityEvent to refresh automatically.

This approach mirrors the ScriptableObject-driven architecture proposed by Ruiz Rabasseda [9], in which ScriptableObjects serve as the foundation for a Unity development workflow. It also aligns with Unity’s description of ScriptableObjects as reusable asset-based data stores that can be shared across runtime objects. In this project, the approach proved effective

because adding a new item required creating only one asset in the Unity Editor, with no code changes.

IV. GAMEPLAY MECHANICS

A. Survival Stats

The SurvivalStats MonoBehaviour manages two primary stats: Hunger and Energy. Hunger decays at a configurable rate (hungerDecayRate per in-game minute). After the player eats food, a configurable post-eating pause prevents hunger from immediately resuming its drain, rewarding foraging behaviour. Energy decays only when hunger drops below a configurable threshold (50% by default), creating a natural priority ordering: eat first, then manage stamina.

When both hunger and energy simultaneously reach zero, the onDeath UnityEvent fires, triggering the Game Over screen. Death from either stat alone is intentionally prevented the player must be both starving and exhausted, providing time to react to a single failing stat. The resulting screen, shown in Figure 2, displays the failure state and gives the player the option to retry.

Fig. 2. Game Over screen with death reason, day count, and fade-in transition.

B. Resource Gathering and Animation

ResourceNode components are placed on interactable world objects. Each node stores a primary ItemData and an optional secondary ItemData, allowing trees to drop both Wood and Vine simultaneously. When the player presses E near a node, a Coroutine-based ChopRoutine plays: the tree mesh shakes for a configurable duration using sinusoidal oscillation, then falls via quaternion Slerp interpolation. Items are added to the inventory after the fall, and the visual is hidden. A stump mesh optionally appears, and nodes respawn after a configurable regrowthTime.

Tool requirements are enforced at the interaction level. If a ResourceNode requires a Stone Axe and the player lacks one, the interaction prompt changes from '[E] Chop Wood' to 'Need a Stone Axe to chop Wood,' blocking progress until the axe is crafted from stone and vine resources.

C. Crafting System

The CraftingManager validates recipes by iterating each ingredient's ItemData and checking HasItem() on the InventorySystem. On success, RemoveItem() is called for each ingredient and AddItem() delivers the output. The system fires a UnityEvent<CraftingRecipe> on success, enabling decoupled notification logic.

The two primary recipes are: Raft Plank (5x Wood + 2x Stone) and Raft (4x Raft Plank + 3x Vine). Crafting a Raft triggers a screen notification guiding the player to the shore. The RaftSpawner listens to onInventoryChanged and activates the raft 3D model and glowing escape trigger as soon as the Raft item is detected in inventory.

The crafting interface visually communicates recipe requirements and available ingredients to the player. As shown in Figure 3, the craft menu uses live ingredient validation and a context-sensitive button state to indicate whether an item can be crafted.

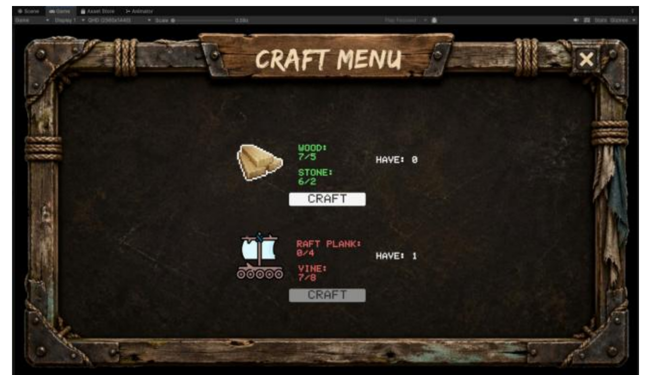

Fig. 3. Craft Menu with live ingredient validation and context-sensitive craft button state.

D. Day/Night Cycle

The DayNightCycle MonoBehaviour advances a timeOfDay float (0–24) using configurable dayDurationSeconds. The sun directional light is rotated proportionally, and its colour and intensity are sampled from AnimationCurve and Gradient. The skybox material's_Exposure and _Tint properties are also driven per-frame to darken the scene at night using the Skybox/Cubemap shader properties. Ambient

light colour and intensity are controlled via RenderSettings, and Dawn, Dusk, and NewDay UnityEvents fire at configurable thresholds.

The effect of this system is visible during night-time gameplay, where the scene becomes darker and the sky exposure changes dynamically. Figure 4 shows the night scene produced by the DayNightCycle system.

Fig. 4. Night scene showing dynamic sky darkening driven by DayNightCycle.

V. UI AND INTERACTION SYSTEMS

A. Input System Integration

The Unity Starter Assets package uses the new Input System with a PlayerInput component and an InputActionAsset. To avoid conflicts, all custom interactions E key for gathering, C key for crafting, Escape for pause were implemented as runtime InputAction objects created in Awake(), bound using the '<Keyboard>/e' path syntax, and enabled/disabled in OnEnable/OnDisable. This pattern requires no InputActionAsset and coexists cleanly with the Starter Assets controller.

B. HUD and Inventory Display

The HUDController subscribes to SurvivalStats.onHungerChanged and onEnergyChanged UnityEvents, updating UI Sliders and their fill Image colours only when values change, avoiding per-frame polling. The InventoryHUD maintains fixed ItemSlotUI entries, each tracking one ItemData asset. Refreshes are triggered exclusively by the InventorySystem.onInventoryChanged event, ensuring the UI is always consistent with the game state. The in-game HUD presents the player's survival information, collected resources, and current day/time information in a compact layout. The final HUD implementation is shown in Figure 5.

Fig. 5. In-game HUD showing hunger/energy bars, resource inventory, and day-time display.

C. Exclusive Panel Management

An ExclusivePanel component, placed on each major UI panel (pause, crafting, game over, ending), closes three sibling panels in its OnEnable callback and restores Time.timeScale and cursor state in OnDisable. This gives each panel self-contained lifecycle management without a central UI controller.

D. Notification System

NotificationUI is a singleton MonoBehaviour providing a static Show (message, duration) API. It drives a CanvasGroup alpha fade coroutine for timed on-screen messages. It is called when a raft is crafted, when a chest is opened listing received loot, and when tool requirements block an interaction.

VI. RESULTS AND DISCUSSION

Fig. 6. Escape Complete screen shown when the player boards the raft.

The completed prototype implements a full gameplay loop from startup through resource gathering, crafting, and escape. The main menu (Fig. 1) and HUD (Fig. 2) demonstrate the polished UI layer built on Unity's UI Toolkit. The crafting menu (Fig. 3) shows live ingredient validation with colour-coded feedback. Figures 4 and 5 illustrate the day/night cycle and the victory screen, respectively. Figure 6 shows the game-over outcome.

The modular architecture proved highly maintainable: stone and berry resource nodes were added without any script changes, requiring only scene objects with ResourceNode components configured in the Inspector. The ScriptableObject recipe system allowed the full crafting chain to be authored without code.

The primary technical challenge was the conflict between Unity's new Input System and the legacy Input.GetKeyDown API, which raises an InvalidOperationException when the Input System package is active. This was resolved by using InputAction objects with performed callbacks exclusively, avoiding the legacy API entirely.

A secondary challenge was event ordering in the game over sequence. Setting Time.timeScale = 0 before StartCoroutine prevented the faded coroutine from advancing since Unity does not process coroutines when time is frozen. This was resolved by starting the coroutine first and using Time.unscaledDeltaTime for all fade calculations. The GameOverPanel also subscribes to SurvivalStats.onDeath in Awake () rather than Start() to guarantee the subscription is registered before any possible death event.

VII. CONCLUSION

This paper presented Survivor's Journey, a Unity 6 survival game built with a modular, event-driven C# architecture. The project demonstrates a complete game loop including survival stats, resource gathering with procedural animation, ScriptableObject-driven crafting, dynamic day/night rendering, and a layered UI system.

Limitations include the absence of enemy AI, a single escape route, and a static island with no procedural generation. Future work will explore NavMesh-based animal AI, a fog-of-war minimap using render textures, a save/load system serialised to JSON, and weather events that create emergent risk/reward decisions. The architectural patterns established particularly the event-driven inventory and the ScriptableObject data model will scale cleanly to these additions.

REFERENCES

1. Strannegård, C., Engsner, N., Ulfsbäcker, S., Andreasson, S., Endler, J., & Nordgren, A. (2024). Survival games for humans and machines. *Cognitive Systems Research*, *86*, 101235.
2. May, L. (2025). Emergent ecological dynamics in videogames: What player paratexts reveal. *Eludamos: Journal for Computer Game Culture*, *16*(2), 161-183. https://doi.org/10.7557/ejcgc.v16i2.7849
3. Anto, A., Basu, A., Selim, R., Foscht, T., & Eisingerich, A. B. (2024). Open-world games' affordance of cognitive escapism, relaxation, and mental well-being among postgraduate students: mixed methods study. *Journal of Medical Internet Research*, *26*, e63760.
4. Filippas, A., & Xinogalos, S. (2023). Elementium: design and pilot evaluation of a serious game for familiarizing players with basic chemistry. *Education and Information Technologies*, *28*(11), 14721-14746.
5. Моісеєнко, Н. В., Моісеєнко, М. В., Кузнецов, В. С., Ростальний, Б. А., & Ків, А. Ю. (2023, September). Teaching computer game development with Unity engine: a case study. CEUR Workshop Proceedings.
6. Gazis, A., & Katsiri, E. (2023). Serious games in digital gaming: A comprehensive review of applications, game engines and advancements. *arXiv preprint arXiv:2311.03384*.
7. Chebanyuk, O. (2023, December). Approach to Reuse of Unity Game Scenes. In *2023 International Conference on Advanced Enterprise Information System (AEIS)* (pp. 11-15). IEEE.
8. Jackson, K. M., & Rosero, A. (2025, September). A Simple Framework to Guide the Development of Online Experimental Cognitive Tasks Using the Unity Game Engine. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* (Vol. 69, No. 1, pp. 119-123). Sage CA: Los Angeles, CA: SAGE Publications.
9. Ruiz Rabasseda, A. (2024). Creating a unity framework for Scriptable Object Driven Development (SODD).