Marwan Charafeddine March 6, 2025

Techniques and Technologies

My Past Projects

This is a summary of some of the work accomplished in the past year and a half across **front-end gameplay programming and much more–**primarily within **Unreal Engine 5** and **Unity**–with a focus on creating immersive and functional game experiences. The following provides an overview of the **technologies, techniques, and implementations** that have been applied to my past projects.

---

1. Front-End Technologies & Techniques

Game Engines & Frameworks

Development has been carried out in:

- **Unreal Engine 5** (C++, Blueprints, Paper2D) for both **2D and 3D projects**.
- **Unity** (C#) for **first-person experiences and physics-based mechanics**.

Examples from Past Projects:

- **2D platforming mechanics** were developed in **Unreal Engine 5.3** using various methodologies.
- A **first-person experience** was created in **Unity**, incorporating movement mechanics and interactive elements on a **lunar-themed level**.
- A **3D Wild West-themed alleyway level** was built in **Unreal Engine 5**, designed with **environmental storytelling and level flow**, taking some aesthetic inspirations from *Red Dead Redemption 2*.

2D Platforming Mechanics in Unreal Engine 5.3 (Paper2D)

A **2D platformer prototype** was developed using **Paper2D** in **Unreal Engine 5.3**.
Key Features Implemented:

- **Character Movement System:**
  - A **Sprite Flipbook Animation** was linked to movement input.
  - InputAxis MoveRight controlled left/right movement, modifying the **velocity** of the CharacterMovement component.
  - Jumping was handled with an InputAction Jump, using **flipbook state changes**.

🔹 **Blueprint Flow:**

```
InputAction Jump → Play Jump Flipbook → Jump
   On Landed → Set Flipbook (Idle/Walk based on velocity)
```

- Character movement was **successfully implemented and tested** with proper animations.

First-Person Movement System in Unity

A **first-person movement system** was created in Unity, using **C# scripts** for movement and camera control.
Key Features Implemented:

- **WASD-based movement**
- **Mouse-controlled camera rotation**
- **Physics interactions with objects**

🔹 **C# Code for Movement:**

```
public class PlayerMovement : MonoBehaviour
{
    public float moveSpeed = 5f;
    public float mouseSensitivity = 2f;
    private CharacterController controller;
    private Vector3 moveDirection;
    private float rotationX = 0f;

    void Start()
    {
        controller = GetComponent<CharacterController>();
        Cursor.lockState = CursorLockMode.Locked;
    }

    void Update()
    {
        // Player movement
        float moveX = Input.GetAxis("Horizontal") * moveSpeed;
        float moveZ = Input.GetAxis("Vertical") * moveSpeed;
        moveDirection = transform.right * moveX + transform.forward * moveZ;
        controller.Move(moveDirection * Time.deltaTime);

        // Mouse look
        float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity;
        float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity;
        rotationX -= mouseY;
        rotationX = Mathf.Clamp(rotationX, -90f, 90f);
        transform.Rotate(Vector3.up * mouseX);
        Camera.main.transform.localRotation = Quaternion.Euler(rotationX, 0f, 0f);
    }
}
```

- The **player successfully moved** using WASD.
- **Mouse rotation worked as expected** with a smooth first-person experience.

Rendering & Graphics

Rendering techniques were implemented to optimize visuals, including:

- **Nanite & Lumen in Unreal Engine 5** to support high-performance rendering for a **detailed 3D environment**.
- **Sprite-based 2D rendering in Paper2D**, utilized in a platformer project.
- **Work on lighting and shadows in both UE5 and Unity**, where **Point and Spot lights were used to simulate a lantern effect**.

Examples from Past Projects:

- **Dynamic lighting and shadows** were set up for **a lunar-themed level in Unity**.
- **Lumen in Unreal Engine 5** was leveraged to create **realistic lighting for a 3D environment**.
- **2D sprite flipbooks** were designed to animate elements within a platformer.

Dynamic Lighting & Shadows in Unity (Lunar-Themed Level)

A **lunar-themed level** was created with **realistic lighting effects** in Unity.
Key Features:

- **Point & Spot Lights** simulated a **lantern effect**.
- **Real-time shadows** were applied.
- **Ambient lighting** was adjusted for a **moonlit atmosphere**.

🔹 **Implementation:**

- A **Spotlight** was attached to the **lantern object**.
- **Shadow intensity** was adjusted in Unity's **Lighting settings**.
- **Color temperature** was fine-tuned to create a cold, **low-gravity lunar aesthetic**.

Gameplay Logic & Mechanics

A variety of **game mechanics** have been designed and implemented, including:

- **Player movement, dashing mechanics, a persistent inventory system and more** in Unreal Engine 5 with Blueprints and C++.
- **First-person movement in Unity**, along with physics-based interactions.
- **A 'game of chance' based progression system**, where various doors determine different player destinations in a **2D platformer** demo project.

Examples from Past Projects:

- **A pause menu** was created in **Unreal Engine 5**, featuring UI buttons for **Pause, Resume, and Quit**.
- **A Blueprint system for throwing dagger projectiles** was implemented in **a 2D action game**.
- **Jump logic and player movement using Unreal Flipbooks** were set up.
- **A ladder climbing mechanic in Unreal Engine** was built using **InputAxis events**.

Inventory & Chest System in Unreal Engine 5 (Mysteries of Tupni)

A **fully functional inventory system** was developed in **Unreal Engine 5**, allowing the player to **collect, store, and retrieve items** from an inventory UI. A **chest system** was also implemented, enabling items to be stored in chests and retrieved later. This system was built using **Blueprints** and involved **UI interaction, data storage, and gameplay logic integration**.

Blueprint names used here are generic, not the actual designations in the project files, as this is a general description of the process used to build these assets/mechanics.

Key Features Implemented:
-Item pickup and storage in an inventory array
-Drag-and-drop inventory system with stackable items
-Separate chest inventories that persisted items
-Dynamic UI updating based on inventory changes
-Interaction system allowing players to open/close chests and transfer items

Inventory System Implementation

Item Pickup & Storage

A **Base Item Blueprint (BP_Item)** was created to represent **collectible items**.

- Each item had **a unique name, an icon, and a quantity variable**.
- A **collision box** detected when the player overlapped an item.
- Items were stored in **an inventory array in the player character blueprint**.

🔹 **Blueprint Logic for Picking Up Items**

On Begin Overlap (Item) → Check Inventory for Existing Item

| If Item Exists → Increase Stack Count

| Else → Add Item to Inventory Array

Destroy Item Actor from World

How It Worked:

- When an item was touched, **its data was checked against the player's inventory array**.
- If the item already existed, **its quantity was increased**.
- If it was a **new item**, it was **added to an inventory array** stored in the player character.
- The **item was then removed from the world**, preventing duplicates.

Items successfully stacked, and inventory was properly updated.

Inventory UI System

An **Inventory Widget (WBP_Inventory)** was created to display collected items.

- **A Grid Panel** dynamically spawned item slots based on inventory contents.
- **Drag-and-drop functionality** was added to allow **manual reordering of items**.
- **Clicking on an item displayed its description and use options**.

🔹 **Blueprint Logic for UI Updating:**

On Inventory Update → Clear Grid Panel → Loop through Inventory Array

| Spawn Item Slot Widgets

| Assign Item Data (Icon, Quantity, Tooltip)

| Bind Click Event for Item Actions

How It Worked:

- When an item was picked up, **the inventory array was refreshed**.
- The UI **removed all existing item slots** and **recreated them dynamically**.
- Clicking an item **opened a context menu**, allowing it to be **used, dropped, or transferred**.

The UI dynamically updated and correctly displayed item quantities.

Chest Blueprint (BP_Chest)

A **Blueprint Actor (BP_Chest)** was created to function as an **item container**.

- **A separate inventory array** was assigned to each chest instance.
- A **collision box** allowed the player to interact with the chest.
- Opening the chest **switched the UI to show chest contents** instead of the player's inventory.

🔹 **Blueprint Logic for Chest Interaction:**

On Player Interact (E) → Open Chest Inventory UI

| Pull Chest's Inventory Array into UI

| Allow Item Transfers (Drag & Drop Between Panels)

How It Worked:

- Pressing **"E" while near a chest** opened its inventory.
- The chest's **inventory array was passed to the UI**, allowing items to be stored or retrieved.
- Items could be **dragged from the player's inventory into the chest** and vice versa.

Items persisted in the chest even when closed and reopened.

Item Transfer Between Inventory & Chest

A **drag-and-drop transfer system** allowed items to be moved between the **player's inventory** and a **chest's inventory**.

🔹 **Blueprint Logic for Drag-and-Drop Transfers:**

On Item Dragged → Detect Drop Target

| If Target is Chest Panel → Move Item to Chest Inventory

| If Target is Player Panel → Move Item to Player Inventory

Update Both Inventory Arrays & Refresh UI

How It Worked:

- Items were **removed from the source inventory and added to the target inventory**.
- The **UI was updated dynamically** to reflect the changes.
- Items **persisted within chests**, meaning they could be retrieved later.

Items correctly transferred and persisted in both inventories.

Additional Features

Item Descriptions & Tooltips

- Hovering over and right clicking an item displays its **name and description**.

Dropping Items from Inventory

- Items could be removed from the inventory and **respawned in the world**.

All Features Were Confirmed Working: Players could **pick up, store, transfer, and drop items** seamlessly.

Multiple inventory arrays handled for both player and chests
Drag-and-drop mechanics successfully implemented
Persistent storage of items within chests
Dynamic UI updates based on inventory changes
Stacking, splitting, and tooltip systems integrated

This **inventory and chest system** was one of the most **complex and polished mechanics implemented in past projects**, requiring **multiple Blueprint interactions, UI handling, and inventory logic**.

UI & User Interface Systems

Work has been done with:

- **UMG (Unreal Motion Graphics) in Unreal Engine**, handling **menus, HUDs, and interactive elements**.

- **Unity UI Canvas**, used for **interactive menus and button-based navigation**.
- **HUD elements** that **updated dynamically based on player interactions**.

Examples from Past Projects:

- **A main menu featuring Start, Quit, and Credits buttons** was designed in **Unreal Engine 5**.
- **An on-screen dice roll indicator** was implemented for **door interactions in a platformer**.
- **A pause menu overlay** was created, allowing the game to be resumed or exited.

Slot-Based Inventory System (Mysteries of Tupni – Unreal Engine 5)

A **fully functional inventory system** allowed players to **collect, store, and use items** with drag-and-drop mechanics. Described in depth in the previous section.

Key Features:
-**Item slots dynamically updated based on player inventory.**
-**Items could be dragged, stacked, and placed in containers (e.g., chests).**
-**Contextual item descriptions appeared on hover.**
-**Interaction with world objects (e.g., keys unlocking doors) was fully integrated.**

🔹 **Blueprint Logic for Inventory System:**

Examples
On Item Pickup → Add to Inventory Array → Update UI
On Item Hover → Display Name & Description
On Item Dragged → Allow Placement in Open Containers
On Item Used (E.g., Key on Door) → Check Item ID → Trigger Unlock Event

Outcome:
-The **inventory UI functioned smoothly**, updating dynamically as items were collected or used.
-The **drag-and-drop system worked as designed**, allowing intuitive item management.

Physics & Animation

Physics and animation systems were integrated into various projects, including:

- **Physics-based interactions in Unity**, utilizing **Rigidbody components**.
- **Animation Blueprints and Flipbooks in Unreal Engine**, applied to character animations.
- **Blueprint-driven environmental hazards**, affecting player movement and interactions.

Examples from Past Projects:

- **A bounce pad mechanic** was built in **Unity**, featuring **a separate physics-based script**.
- **A jumping spider enemy with randomized movement patterns** was developed in **Unreal Engine**.
- **A 2D spike trap in Unreal Engine**, which reset the level upon player contact, was implemented.

Dynamic Ragdoll-Driven Player Movement (Ragdoll Plainly Perilous – Unreal Engine 5)

Unlike traditional player movement, the **Chaos Physics Engine** was used to **fully control the player via 'ragdoll' mechanics**.

Key Features:
-**Player movement was purely physics-driven, creating chaotic but fun interactions.**
-**Forces applied dynamically based on player input.**

Outcome:
-Movement remained dynamic and unpredictable, making survival engaging.

Networking & Multiplayer (Client-Side)

Although **single-player mechanics** have been the focus, networking concepts have been explored, including:

- **Basic client-side interactions**, such as **player movement and item replication**.
- **Steamworks API**, integrated for **multiplayer features**.

---

2. Back-End (Server-Side) Technologies & Techniques

Data Persistence & Game State Management

While most projects have focused on **single-player mechanics**, some backend elements have been considered, including:

- **Saving and loading game states** using **Blueprints or C++ in Unreal Engine**.
- **Storing player progress** in the form of **inventory state**, **player level**, **NPC states, etc.**

Examples from Past Projects:

- **Basic save/load mechanics for player progress** were implemented in **Unreal Engine**.

---

3. DevOps in Game Development

Build & Deployment Automation

Stand-alone project builds have been created using:

- **Unreal Engine's project packaging tools**.
- **Unity's build settings for Windows-based exports**.

Server & Performance Optimization

- **Reducing sprite overdraw and optimizing textures** for **2D games**.
- **Cutting out unnecessary** sections of code or Blueprints.
- **Utilizing occlusion culling techniques** in **Unreal Engine 5** to improve performance in **3D scenes**.

Examples from Past Projects:

- **Optimized level streaming for a large 3D environment** in **Unreal Engine 5**.

---

4. Summary of Technologies Used in Past Projects

Category	Technologies Used	Example Implementations
Game Engines	Unreal Engine 5 (C++, Blueprints), Unity (C#)	2D platformers, first-person Unity experience, Wild West UE5 level, 2D infinite scrolling SHMUP in UE5, 2.5D infinite survival game
Rendering & Graphics	Nanite (UE5), Lumen (UE5), Unity URP	Realistic 3D environments, 2D sprite rendering
Gameplay Logic	Blueprints, C++, C#	Player movement, interactive elements (buttons, doors, etc.), audio implementation, UI and HUD widgets
UI/UX	UMG (UE5), Unity Canvas	Main menus, pause menus, HUD elements
Physics & Animation	Flipbooks (UE5), Rigidbody (Unity) Chaos (UE5)	Bounce pads, various enemy AI, Ragdoll
Networking (Basic)	Steamworks API	Testing with client-side replication
DevOps (Basic)	UE5 Packaging, Unity Build Settings	Creating standalone project builds
Optimization	Occlusion culling, level streaming, physics optimizations	Performance improvements for large-scale levels, performance improvements within C# scripts and Blueprints by cutting unnecessary or redundant code