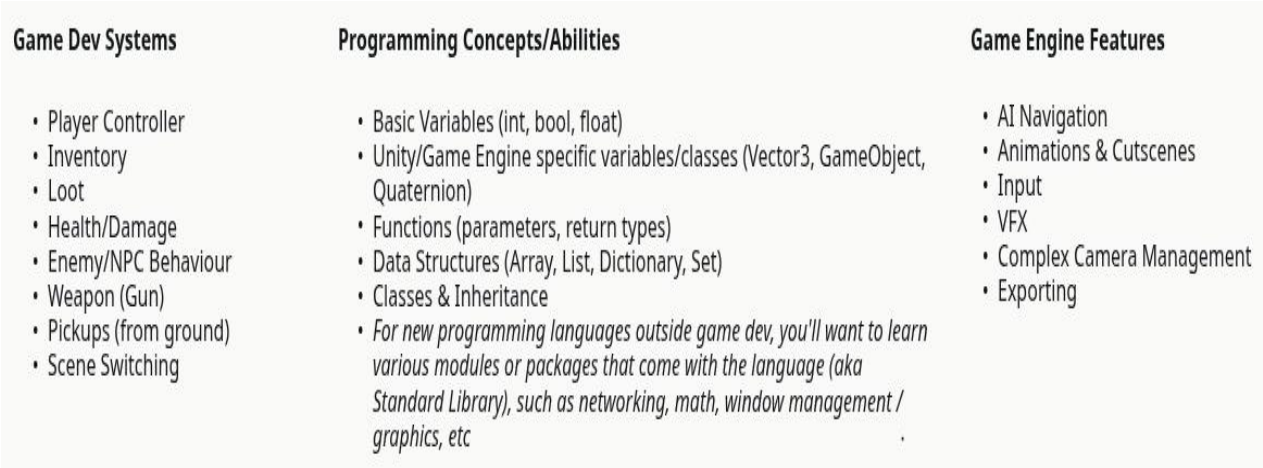
Andrew Mackay
Athabasca University

Toyz Cars UEFN Final

Within this document, I will highlight several concepts I have learned about gameplay development in general and features within the UEFN engine, while also covering several ideas I would work on implementing if I had more time.

1. Core Gameplay Development Approach



Game Dev Systems	Programming Concepts/Abilities	Game Engine Features
• Player Controller • Inventory • Loot • Health/Damage • Enemy/NPC Behaviour • Weapon (Gun) • Pickups (from ground) • Scene Switching	• Basic Variables (int, bool, float) • Unity/Game Engine specific variables/classes (Vector3, GameObject, Quaternion) • Functions (parameters, return types) • Data Structures (Array, List, Dictionary, Set) • Classes & Inheritance • *For new programming languages outside game dev, you'll want to learn various modules or packages that come with the language (aka Standard Library), such as networking, math, window management / graphics, etc*	• AI Navigation • Animations & Cutscenes • Input • VFX • Complex Camera Management • Exporting

Above are notes I took from a session with one of my gameplay tutors. Gameplay development becomes clearer when you can break down the systems that define the gameplay experience. Going a step deeper are the specifics needed to carry out features of these systems by means of programming concepts and abilities, defining this step will help others upskill where needed in order to implement the required programming logic for new features which is not engine specific, although unity is referenced in this specific instance. Breaking down game engine features helps clear the fog from an otherwise daunting task initially of knowing where to start when approaching game engines.

It is helpful to follow along tutorials which outline any of these specific systems in any engine, as some implementations will differ but the core concepts will be transferable. Ultimately when designing features this image can be used as a reference to help frame and conceptualize further clarity for course of action. First what game system does the feature belong. Then what functions/ what variables do I need to make the feature? What game engine features are in use or interact with by these functions and variables?

2. UEFN Engine features background, and Hackathon implementations

It is key to recognize some distinct differences between Git version Control and UEFN version control to avoid confusion.

UEFN's version control system is specifically designed to work within the context of Unreal Engine projects, offering several unique features compared to traditional Git:

Automatic Asset Locking: When an asset is being edited, it is automatically checked out and locked, preventing others from making conflicting changes at the same time.

In-Editor Workflow: Unlike Git, which is primarily managed through external tools, UEFN integrates version control directly into the editor, making the process seamless and less disruptive.

Syncing with Fortnite Creative: UEFN allows developers to synchronize changes directly with Fortnite Creative mode, ensuring live updates without manual intervention.

Conflict Resolution: UEFN offers built-in tools to help prevent and resolve conflicts, such as auto-reverting to prevent overwriting changes.

-“https://dev.epicgames.com/documentation/en-us/uefn/unreal-revision-control-in-unreal-editor-for-fortnite"

Related to the version control experience, is testing the game through the launch game button. You will need to make sure your local changes are pushed locally before testing the game within another window which will load separately to the editor.

In the **Toyz Hackathon** Wole and I, managed to succesfully implement the racing experience following this tutorial “https://dev.epicgames.com/documentation/en-us/uefn/build-a-carracing-game-in-unreal-editor-for-fortnite” within the UEFN editor having first built a small map layout after uploading various Toyz assets from Unity.

The game succesfully works within the environment, however when testing the environment the roads were added to be slightly too small for the speed and size of the vehicles, with borders being removed for timeliness of testing. These features will have to be improved and expanded upon for more seamless playability.

Tying it back to general game programming concepts above, it is important to note that the core game mechanics should be developed first to ensure the game is testable before focusing on artistic elements, as this takes away valuable time and can even break the project before actually being able to test it.

- “https://www.quora.com/An-experienced-game-developer-asked-me-to-focus-on-gameplay-first-before-the-graphics-but-how-do-we-get-interested-in-developing-if-all-we-see-are-basic-meshes”

This was the case for our approach in the Hackathon as an imported asset from Unity had a ‘+’ symbol which defied the naming conventions in the UEFN engine. This

error had us stuck on this error for hours, partly because in other engines would likely still be able to playtest the game but with this one asset being altered, however in this case it caused the game to be unplayable.

3. Transporting characters after the race with cutscenes

Something we had planned on doing but had ran out of time was to be able to have players transported from one building onto others through using the UEFN Teleporter device. The idea was to provide the players with post game chat areas which they could free roam in an open world.
To implement teleportation in UEFN using the Teleporter device, you would need to first place the Teleporter device in the map and configure its properties, such as its target location (destination). The player interacting with the Teleporter will then be transported to the destination location. For post-game chat areas, you can set up multiple teleporters in different locations of the open-world area where players can freely roam. Additional customization options can include triggers for teleportation and the creation of seamless transitions between different environments. - https://dev.epicgames.com/documentation/en-us/uefn/unreal-revision-control-in-unreal-editor-for-fortnite.

Also another feature was to implement a rain weather system which was triggerable by the driver hitting a spider or crab-bot and this then changing the weather or game mode, upon when the race is finished and the bots which represent self defeating thoughts are defeated then the weather would change back to sunny. We had experimented with a few different approaches to achieve this but ultimately ran out of time in the process and what is in place is only half working.

```

1. Setting Up the Weather System:
Create Weather Effects: Utilize UEFN's Niagara system to design custom weather effects like rain. The official documentation provides guidance on creating effects and particle systems:
https://dev.epicgames.com/documentation/en-us/uefn/effects-and-particle-systems-in-unreal-editor-for-fortnite

Implement Weather Transitions: Develop a Verse script that manages weather transitions based on game events. For instance, when a player interacts with a specific bot, the script can trigger a change from sunny to rainy conditions.
https://dev.epicgames.com/documentation/en-us/uefn/effects-and-particle-systems-in-unreal-editor-for-fortnite

2. Detecting Player Interaction with Bots:
Place Trigger Devices: Use UEFN's trigger devices to detect player interactions. When a player interacts with a bot (e.g., a spider or crab-bot), the trigger device can send a signal to initiate the weather change.
https://dev.epicgames.com/documentation/en-us/uefn/coding-device-interactions-in-verse
```

Script Event Handling: Write Verse code to handle events from these trigger devices. For example, subscribing to the interaction event of a trigger device can execute a function that changes the weather:

Example code -->
using { /Fortnite.com/Devices }

myTriggerDevice := class(creative_device):

    Trigger: trigger_device = trigger_device{}
    OnBegin<override>()<suspends>: void =
        Trigger.InteractedWithEvent.Subscribe(OnInteracted)

    OnInteracted(Agent: agent): void =
        # Code to change weather to rain
        ChangeWeather("rain")
https://dev.epicgames.com/documentation/en-us/uefn/coding-device-interactions-in-verse

3. Reverting Weather After Race Completion:
Monitor Race Status: Implement a system to track the race's progress and determine when it finishes. This could involve setting up game mode variables or using UEFN's game state management features. As the race trigger has already been sucessfully implemented it should be easy to locate and access.

https://dev.epicgames.com/documentation/en-us/uefn/coding-device-interactions-in-verse

Reset Weather: Once the race is finished and the bots (representing self-defeating thoughts) are defeated, trigger a weather change back to sunny conditions using a similar Verse script:

Pseudo code sample -->
Function to change weather
ChangeWeather(WeatherType: string): void =
    if WeatherType == "rain":
        # Code to activate rain effect
    else if WeatherType == "sunny":
    ```

## 4. Deep linking to Rocket Racer from Unreal Fortnite

Deep Linking in Unreal Editor for Fortnite (UEFN)
Unreal Editor for Fortnite (UEFN) only offers partial support for deep linking, a feature that enables users to open specific in-app content via URLs. Epic Games has introduced special URLs that let creators share direct access to their published Fortnite islands. These links, when clicked, launch Fortnite and navigate the user to
    ```

the designated island through the game's home bar, enhancing content discoverability and user access (Epic Games, 2024).

Currently, this functionality is intended for external platforms such as websites, social media, and marketing channels. As of early 2025, there's no support for triggering deep links internally within Fortnite using Verse scripts or other in-game systems. This means developers cannot link from one island to another during gameplay via deep linking. Instead, the feature acts solely as an external access method.

There have also been technical setbacks. Notably, in March 2025, Epic Games temporarily suspended deep linking on PC due to unresolved issues (Epic Games Forums, 2025). Despite its utility for driving external traffic, deep linking lacks functionality for in-game navigation.

In conclusion, while UEFN's deep linking facilitates external promotion and access, it's not yet integrated for in-game use or automation via scripting. Developers must continue to use UEFN's standard mechanics—like teleporters and in-game devices—for internal navigation between islands.

References

Epic Games. (2024, August 10). Instantly transport players to your Fortnite island with a unique URL. Retrieved from https://create.fortnite.com/news/instantly-transport-players-to-your-fortnite-island-with-a-unique-url

Epic Games Forums. (2025, March). Deep linking temporarily disabled on PC. Retrieved from https://forums.unrealengine.com/t/deep-linking-temporarily-disabled-on-pc/2375350

Proposal to Epic to allow deep linking through UEFN to Rocket Racer on behalf of Toyz

Hi Epic Team,

I'd like to propose enabling an in-game deep linking feature from Verse UEFN to Unreal Rocket Racer. While current deep links support external redirection to islands, there's no way to programmatically trigger them from within a live session. Adding this feature would allow developers to create seamless transitions between islands based on player actions, unlocking new use cases for multi-island experiences and modular game design. It would greatly help the Toyz Electronics Racer game be the top tier racing experience it has the potential to be.

Thanks for considering this idea!

Sincerely,
Andrew Mackay

Here is a draft of what this code would look like-->

```
using { /Verse/Log } as Log
using { /FortniteDevices } as FortniteDevices

class DeepLinkSimulatorDevice := class(FortniteDevices.CreativeDevice):

    # Editable reference to a teleporter device placed in the UEFN editor
    @editable
    Teleporter: FortniteDevices.teleporter_device = FortniteDevices.teleporter_device{}

    # Called when the device is initialized in the world
    OnBegin := function(): void =
        Log.log("DeepLinkSimulator initialized. Awaiting simulated deep link...")

    # This simulates receiving a deep link
    HandleDeepLink := function(island_code: string): void =
        if (island_code != ""):
            Log.log("Deep link received with island code: " + island_code)
            TeleportPlayers()
        else:
            Log.log("No island code provided. Aborting teleportation.")

    # Teleports all players in the current playspace using the linked Teleporter device
    TeleportPlayers := function(): void =
        players := GetPlayspace().GetPlayers()
        for (player in players):
            Teleporter.Teleport(player)
            Log.log("Teleported player: " + player.GetName())
```

5. Set up basic cinematics

Zooming in on the Deity Dah character in the horizon after a key event was an idea that we wanted to implement but also did not have time. Using these references I can infer a general guide to completing this task.

-UEFN Sequencer (official doc):
“https://dev.epicgames.com/documentation/en-us/uefn/sequencer-overview”
-Camera Work & Cutscenes with Devices:
“https://dev.epicgames.com/documentation/en-us/fortnite-creative/using-cinematic-devices”
-Verse to control cutscenes:
“https://dev.epicgames.com/documentation/en-us/uefn/verse-api/devices/cinematic_sequence_device”

To create a dramatic character reveal cutscene in UEFN, start by setting up your environment where the encounter will happen. Design the area with moody lighting, obstacles/objects, and maybe some animated effects like fog, glowing runes, or energy pulses to build atmosphere. Place the character in a hidden state—either underground, behind a door, or invisible—so that it only appears during the cinematic.

Next, add a Cinematic Sequencer device into the level. This will be responsible for playing the cutscene. In the World Outliner, create a new Level Sequence asset (e.g., "CharacterRevealSequence") and open it to begin editing the timeline. Add a Cinematic Camera Actor and pilot it to design your shots. A good reveal usually starts with a wide environmental shot to set the mood, then cuts to closer angles showing environmental changes (like torches lighting up or rocks falling), and finally ends on a slow upward tilt or zoom onto the boss as it rises, turns, or roars.

Inside the Level Sequence, animate the character entering the scene, in this instance pondering on a building top in the distance. Animate the camera to slowly move around the character. Add keyframes to control visibility of the boss mesh if it should appear mid-sequence. For extra flair, add sound cues like ambient music fading into dramatic combat music, or play some sort of voiceover line from the boss.

Once the sequence is complete, assign it to the Cinematic Sequencer device in the level. Then, place a Trigger Device near the entrance to the boss arena. Set it to activate when a player enters the trigger zone and configure it to send a signal to the Cinematic Sequencer. In the Sequencer's settings, make sure it's set to play when triggered. If needed, you can use a Player Teleporter to reposition the player in the best spot for the cutscene before it starts.

For additional control, you can write a short Verse script that listens for the player entering the boss arena and then plays the cinematic. The Verse script could also disable player controls temporarily during the cutscene and reactivate them when it ends, creating a clean transition into the boss battle. After the cutscene finishes, Verse can trigger the boss's AI activation or start the combat phase.

Finally, test the encounter by entering the designated cutscene zone. Ensure the cutscene plays smoothly, the boss animates as expected, and the transition to gameplay is seamless.

Cutscenes can also be triggered by Verse code like such:

```
using { /Fortnite.com/Devices }

cutsceneTrigger := class(creative_device):
    @editable
    Trigger: trigger_device = trigger_device{}
    Cinematic: cinematic_sequence_device = cinematic_sequence_device{}

    OnBegin<override>()<suspends>: void =
        Trigger.TriggeredEvent.Subscribe(OnTriggered)
```

```
    OnTriggered(Player: agent): void =
        Cinematic.Play()
```

Further considerations

Keeping in mind that transferability of player data created within Unity will likely not transfer at all between Unity to UEFN, it is best to think of this experience as separate. If avatars are necessary UEFN has their own avatar creation system but keep in mind you will likely lose players attention if they have to recreate an avatar just to play another aspect of your game.

Microsofts notes for the unity racer game indicates they want to cut to playable core game before doing any additional features which may deepen the experience if arranged properly, however distract from the experience if not. I found these notes interesting and would likely take them into consideration for the UEFN racer as well.