FPS Game Project

In-depth weapon system

Colman Nicolas

Email: nicolman01@gmail.com

Linkedin: https://www.linkedin.com/in/nicolas-colman-27423b2b2/

Github: https://github.com/NicoColman

Index

- Introduction

Welcome to an in-depth exploration of the innovative weapon creation workflow behind our latest First-Person Shooter (FPS) game, developed with Unreal Engine 5. This document is designed to offer insights into the technical and creative processes that underpin one of the game's most dynamic and engaging features—its arsenal of unique, customizable weapons. As you journey through these pages, you will gain a comprehensive understanding of the methodologies, technologies, and design principles that have been meticulously combined to bring our game's weaponry to life.

Purpose:

This document serves not only as a showcase of what has been accomplished in our latest project but also as a testament to the passion, expertise, and creativity that drive our game development endeavors. Whether you are a fellow developer, a potential collaborator, or simply a fan of the gaming world, our aim is to provide you with a fascinating glimpse into the making of a game that pushes boundaries and sets new standards for immersive gameplay.

2. Technical Details:

Engine Used:

The game was developed using Unreal Engine 5, leveraging its cutting-edge graphics and physics capabilities to deliver a visually stunning and immersive gameplay experience. Significant modifications and proprietary technologies were developed to tailor the engine to our specific needs, ensuring a unique gameplay experience.

Programming Languages:

The core game logic and functionality were implemented using C++, providing the performance and flexibility needed for a high-fidelity FPS game. Unreal Engine's Blueprints were also utilized for rapid prototyping and to implement certain game mechanics, allowing for a more iterative and visual

approach to development. Additionally, Python scripts were developed for automating the loading of weapon assets and properties, streamlining the asset pipeline, and ensuring consistency across the game.

Data Handling:

JSON format was employed for storing all weapon properties, facilitating easy adjustments and scalability. This choice allowed for a more organized and manageable way to handle the game's extensive arsenal, making it easier to tweak and balance gameplay.

Performance Optimizations:

A cornerstone of our optimization strategy was the adept use of TSoftObjectPtr for asynchronous loading and unloading of game assets. This method considerably improved the game's performance by facilitating smooth management of memory and curtailing load times. Through asynchronous asset management, we guaranteed that the game consistently runs at high frame rates and maintains responsive gameplay, even in asset-intensive scenarios. This technique proved invaluable for dynamically managing weapon assets and properties, optimizing resource use without compromising the game's visual fidelity or immersive qualities.

- Innovative Asset Management: To further optimize the game's performance and asset management, we implemented a novel system involving the creation of a Weapon Proxy, or Placeholder. This proxy has no inherent functionality but holds a reference to a UPrimaryDataAsset that contains the actual properties for both first-person and third-person perspectives. These assets are only loaded into the game when the weapon is picked up by the player. This approach significantly reduces the initial load on the game's memory and processing, as it prevents unnecessary pre-loading of all weapon assets.

- Weapon Instantiation Process: The UPrimaryDataAsset plays a critical role beyond just holding weapon properties; it is also utilized within a weapon factory for the instantiation of the actual weapon. This system allows for the efficient creation of weapons on the fly, ensuring that resources are allocated only when necessary and are immediately available when required by the gameplay. This method not only streamlines the weapon handling process but also contributes to a more dynamic and responsive gaming experience.

- Python Scripting for Asset Loading: Leveraging Python scripts for the automated loading of UPrimaryDataAsset references for first-person, third-person, and proxy weapons has been a game-changer. This scripting not only automates a critical aspect of the game's asset management but also ensures a high level of consistency and efficiency in how assets are handled across the development process. The integration of Python scripting underscores our

commitment to using advanced programming techniques to solve complex game development challenges.

3. Design Patterns:

In the development of the game, we employed several key design patterns to ensure the codebase was both efficient and manageable. These patterns facilitated the creation of a robust and flexible architecture, allowing for easy expansion and modification as the game evolved.

Factory Pattern:

We utilized the Factory Pattern for the instantiation of different weapons within the game. This approach enabled us to centralize weapon creation, making it easier to introduce new weapon types and manage their properties in a unified manner. It significantly simplified the process of scaling the game's arsenal while maintaining consistency and reducing the potential for errors.

Observer Pattern:

The Observer Pattern was instrumental in handling player inputs. By decoupling the game's input system from the actions performed in response to those inputs, we achieved a highly responsive and easily modifiable input handling system. This pattern allowed for the dynamic registration and deregistration of events, facilitating a clean and efficient way to update the game state in response to player actions.

Interfaces:

To ensure that different modules of the game remained decoupled, interfaces were extensively used. This approach allowed for communication between modules without requiring them to have direct knowledge of each other, promoting a modular architecture. By defining clear interfaces, we were able to enhance the game's maintainability and flexibility, making it straightforward to add or modify functionalities without affecting unrelated systems.

Singleton Pattern:

The Singleton Pattern was applied in the management of the game's Asset Manager. This ensured that a single, globally accessible instance of the Asset Manager was available, managing all game assets centrally. This pattern was particularly beneficial for asset loading and unloading, providing a consistent access point throughout the game and avoiding the overhead of multiple instances.

These design patterns played a crucial role in structuring the game's codebase, contributing to its performance, scalability, and ease of maintenance. By thoughtfully applying these patterns, we were able to create a game that is not only technically sound but also flexible enough to grow and evolve.

4. Weapon Creation Workflow

Overview:

In the development of our FPS game, one of the cornerstone innovations lies in the weapon creation and management system. This system is designed from the ground up to address two critical aspects of game development: scalability and maintainability. At the heart of this system is a workflow that meticulously manages the lifecycle of weapons, from conception to in-game use, through a highly decoupled and automated process.

The foundation of this workflow is a robust Python scripting mechanism that automates the loading of weapon properties and assets into UPrimaryDataAsset instances. These assets encompass all necessary data for both first-person and third-person weapon representations, including visual models, animations, and gameplay properties. This approach ensures a consistent and efficient process for managing a vast array of weapon assets, laying the groundwork for easy expansion and updates.

Central to our design philosophy is the decoupling of the AWeaponClass from the specifics of individual weapons. AWeaponClass acts as a general class, unaware of the distinct characteristics of the weapons it will initialize. Instead, it dynamically instantiates weapons in-game based on the corresponding UPrimaryDataAsset. This decoupling not only significantly enhances the maintainability of the codebase by segregating the weapon's logic from its representation but also allows for a modular and flexible system that can easily accommodate new weapons or modifications to existing ones without the need for extensive code overhauls.

This workflow culminates in a weapon factory mechanism that leverages the UPrimaryDataAsset references to dynamically create weapon instances as they are required in the game. This just-in-time creation process is pivotal for optimizing resource usage and ensuring that the game environment remains responsive and engaging for the player.

The weapon creation workflow exemplifies a forward-thinking approach to game development, prioritizing efficiency, scalability, and maintainability. It is a testament to our commitment to creating a robust and flexible system that can grow with the game, ensuring that expansion and updates can be integrated seamlessly without compromising on performance or player experience.

Explanation video: https://www.youtube.com/watch?v=W4pPYq8lTTE

Step 1: Python Scripting for Asset Reference Loading

The initial stage of our weapon creation workflow employs Python scripting to automate the loading and management of weapon asset paths, ensuring a streamlined and error-resistant process. This scripting approach is built around a JSON file that serves as the central repository for all asset paths, including those for first-person, third-person, and proxy weapon representations.

Key Advantages:

- Consistency and Backup: By centralizing asset paths in a JSON file, we establish a consistent reference point for all weapon assets. This system also offers a straightforward method for creating backups, allowing for quick recovery and recreation of UPrimaryDataAsset instances if necessary. The JSON structure ensures that asset management remains organized, making the system highly resilient against errors and inconsistencies.

- Efficient Population of PrimaryDataAssets: The Python script is designed to expedite the population of UPrimaryDataAsset instances. Since it operates by loading asset paths from the JSON file, the process bypasses the manual entry of paths, significantly speeding up the preparation of assets for use in the game. This efficiency is crucial for managing a large arsenal of weapons, from melee to firearms to special items, without bogging down development with tedious manual processes.

- Dynamic Loading with Naming Standards: A pivotal feature of our scripting approach is its ability to dynamically load specific weapons based on their names. Thanks to a standardized naming convention for all DataAssets, the script does not require adjustments to the UPrimaryDataAsset path for each weapon. Merely changing the weapon name in the script's parameters automatically directs it to the correct asset paths. This dynamic capability facilitates rapid iteration and the addition of new weapons to the game, ensuring that the weapon-creation process remains both agile and efficient.

- Maintainability: The combination of a centralized asset repository, efficient population methods, and dynamic loading significantly enhances the maintainability of the weapon system. Developers can easily add, modify, or remove weapons without delving into the complexities of the Unreal Engine asset management system. This approach not only saves valuable development time but also ensures that the game can evolve and expand with minimal friction.

- This first step in our weapon creation workflow underscores our commitment to utilizing advanced scripting and data management techniques to enhance game development efficiency. By leveraging Python's scripting capabilities in conjunction with a carefully designed asset management strategy, we have established a foundation that supports rapid development, easy maintenance, and seamless expansion of our game's weaponry.

Code snippet:

fp_firearm.json

```
{
    "WeaponName": "AG14W",
    "WeaponDamage": 20,
    "WeaponFireRate": 0.1,
    "bWeaponAutomatic": true,
    "WeaponAmmo": 30,
    "WeaponMagAmmo": 30,
    "WeaponCarriedAmmo": 120,
    "WeaponMaxAmmo": 150,
    "WeaponMesh": {
        "WeaponSkeletalMesh": "/Game/Assets/Weapons/Firearms/AG14W/Meshes/SK_LPAMG_AG14W"
    },
    "WeaponStaticMeshes": {
        "SOCKET_Magazine": {
            "SocketName": "",
            "Mesh": "/Game/Assets/Weapons/Firearms/AG14W/Meshes/SM_LPAMG_AG14W_Magazine_Default"
        },
        "SOCKET_Magazine_Reserve": {
            "SocketName": "",
            "Mesh": "/Game/Assets/Weapons/Firearms/AG14W/Meshes/SM_LPAMG_AG14W_Magazine_Default"
        },
```

populate_weapon_firearm_data.py

```python
def populate_weapon_data(desired_weapon_name):

    #Populate FP Firearm Data
    base_asset_path = "/Game/Blueprints/DataAssets/Weapons/FPWeapons/Firearms/"
    json_data = load_json_data("fp_firearms_data.json")

    #Populate TP Firearm Data
    #base_asset_path = "/Game/Blueprints/DataAssets/Weapons/TPWeapons/Firearms/"
    #json_data = load_json_data("tp_firearms_data.json")

    weapon_data = find_weapon_data(desired_weapon_name, json_data)

    if weapon_data:
        full_asset_path = f"{base_asset_path}DA_W_FP_FA_{desired_weapon_name}.DA_W_FP_FA_{desired_weapon_name}"
        #full_asset_path = f"{base_asset_path}DA_W_TP_FA_{desired_weapon_name}.DA_W_TP_FA_{desired_weapon_name}"

        weapon_data_asset = unreal.load_object(None, full_asset_path)

        if weapon_data_asset:

            #print(weapon_data) #Debugging

            #Name
            weapon_data_asset.set_editor_property("WeaponName", weapon_data["WeaponName"])

            #Damage
            weapon_data_asset.set_editor_property("WeaponDamage", weapon_data["WeaponDamage"])

            #Fire
            weapon_data_asset.set_editor_property("WeaponFireRate", weapon_data["WeaponFireRate"])

            #bAutomatic
            weapon_data_asset.set_editor_property("bWeaponAutomatic", weapon_data["bWeaponAutomatic"])

            #SkeletalMeshes
            weapon_skeletal_mesh = unreal.Map(unreal.Name, unreal.SkeletalMesh)
            for key, path in weapon_data["WeaponMesh"].items():
                skeletal_mesh = unreal.load_asset(path)
                if skeletal_mesh:
                    weapon_skeletal_mesh[unreal.Name(key)] = skeletal_mesh
                else:
                    print(f"Failed to load skeletal mesh asset: {path}")
            weapon_data_asset.set_editor_property("WeaponMesh", weapon_skeletal_mesh)
```

Unreal Engine 5 script execution:

```
LogPython: Weapon data populated successfully for WAK12!
Cmd: py "../../../../../../Users/nicol/OneDrive/Documentos/Unreal Projects/Zombies/scritps/populate_weapon_fp_firearm_data.py"
LogFileHelpers: All files are already saved.
LogPython: Weapon data populated successfully for AG14W!
LogSlate: Took 0.000265 seconds to synchronously load lazily loaded font '../../../Engine/Content/Slate/Fonts/Roboto-Light.ttf' (167K)
LogWorldSubsystemInput: UEnhancedInputDeveloperSettings::bEnableWorldSubsystem is false, the world subsystem will not be created!
```

Step 2: Weapon Proxy Creation

The second step in our sophisticated weapon creation workflow involves the development of a Weapon Proxy system, designed to minimize memory overhead and streamline the asset loading process within the game environment. This system leverages a two-part approach involving both a Proxy Primary Data Asset and an AProxyWeapon Actor, which together ensure efficient asset management and seamless integration into the game world.

- Creating the Proxy Primary Data Asset:

  Proxy Primary Data Asset Definition: Initially, we create a Proxy Primary Data Asset that contains only the essential elements for weapon representation: the meshes and references to both the First-Person (FP) and Third-Person (TP) Weapon assets. This streamlined Data Asset is crucial for maintaining performance efficiency, as it holds no functional game logic or unnecessary data that would otherwise increase load times and memory usage.

Performance Optimization: The choice to exclude functionality from the Proxy Data Asset is deliberate, aimed at reducing the computational overhead when these proxies are present in the game map but not actively being interacted with by the player. This strategy is particularly effective in environments where numerous weapons are available for pickup, ensuring that the game remains responsive and fluid.

Implementing the AProxyWeapon Actor

AProxyWeapon Actor Components: Following the creation of the Proxy Data Asset, the next step is to implement an AProxyWeapon Actor. This Actor is equipped with all the necessary UActorComponents to facilitate the eventual activation and use of the weapon within the game. These components are designed to be universally compatible with any Actor, not just the proxy, providing a versatile foundation for weapon functionality.

Reference to the Proxy Data Asset: The AProxyWeapon Actor holds a reference to the newly created Proxy Primary Data Asset, linking the visual and referential aspects of the weapon with the functional components required for in-game interaction.

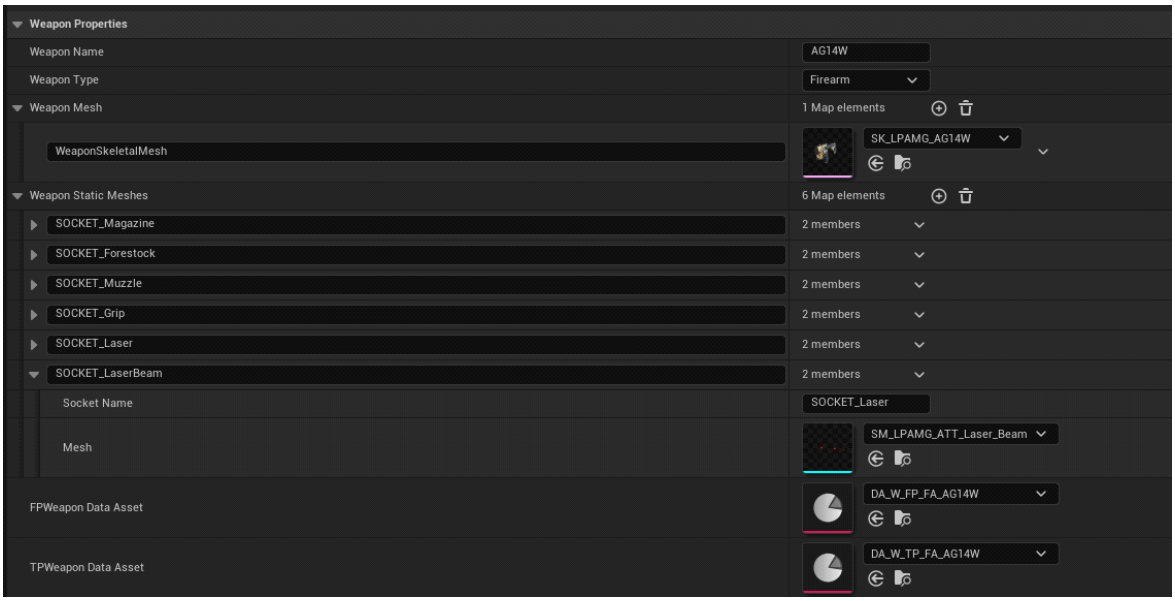
World Integration: Upon completion, the AProxyWeapon Actor can be dragged into the game world for testing and verification. Successful integration is evident when the proxy operates flawlessly within the game environment, visually representing the weapon without imposing any unnecessary load on the system until player interaction necessitates the loading of full weapon assets.

Visual Confirmation and Testing

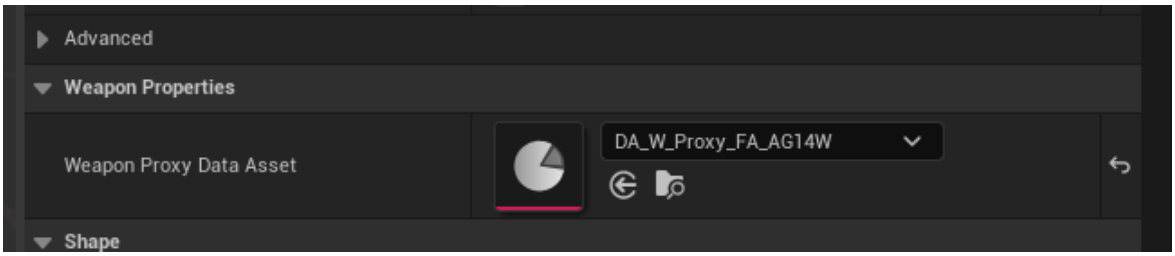
In-Game Verification: The final step in the Weapon Proxy creation process is a thorough in-game testing phase. This phase confirms that the proxy weapons are visually accurate and ready for interaction, without adversely affecting game performance. Testing ensures that when a proxy is approached or activated by the player, the full weapon assets are loaded dynamically, maintaining the game's immersive experience while optimizing resource usage.

This detailed approach to Weapon Proxy creation highlights our commitment to performance optimization and modular design. By carefully separating visual representation from functional logic until the moment of necessity, we achieve a balance between a rich, interactive game world and the technical requirements of high-performance gameplay.

DA_Weapon_Proxy



AWeaponProxy



Step 3: Dynamic Weapon Instantiation via Factory Method

With the AWeaponProxy successfully demonstrating its capability to spawn both FP and TP weapons, we delve into the behind-the-scenes mechanics that power this seamless transition from proxy to functional weapon.

The Factory Method:

- Weapon Type Specification: Within the ProxyDataAsset, we specify the weapon's type (e.g., Firearm, Melee), which informs the factory method about the kind of weapon it needs to create.

This classification is crucial for ensuring that the factory method can accurately instantiate the appropriate weapon class based on gameplay requirements.

- DataAsset Passing: Alongside the weapon type, the data assets for the new weapons are passed to the factory. This step is vital for equipping the instantiated weapon with its specific properties and visual assets. By setting the DataAsset upon creation, each weapon is imbued with the unique characteristics defined within its respective DataAsset.

- Universal Components and Customization: The new weapon classes, like the proxy, are designed with universal components. This architecture ensures that upon setting the specific weapon DataAsset, the weapon automatically adopts the necessary functionalities and attributes, enabling it to operate flawlessly within the game environment.

This factory method not only exemplifies a modular and efficient approach to weapon instantiation but also reinforces the game's scalability and ease of content expansion. By abstracting the weapon creation process through this method, we ensure a system that can dynamically adapt to new weapon types and functionalities, significantly enhancing the game's developmental agility and player experience.

WeaponFactory.cpp

```cpp
AActor* AWeaponFactory::CreateWeapon(EWeaponTypes WeaponType, UPrimaryDataAsset* DataAsset, UWorld* World,
                                     AActor* Actor)
{
    switch (WeaponType)
    {
    case EWeaponTypes::EWT_Proxy:
        return CreateWeaponProxy(World, DataAsset, Actor);
    case EWeaponTypes::EWT_Firearm:
        return CreateFirearm(World, DataAsset);
    case EWeaponTypes::EWT_Melee:
        return CreateMelee(DataAsset);
    default:
        return nullptr;
    }
}

AWeaponProxy* AWeaponFactory::CreateWeaponProxy(UWorld* World, UPrimaryDataAsset* DataAsset, AActor* Actor)
{
    AWeaponProxy* Weapon = World->SpawnActor<AWeaponProxy>(AWeaponProxy::StaticClass(), Actor->GetActorLocation(),
                                                           Actor->GetActorRotation());
    if (DataAsset)
    {
        Weapon->SetDataAsset(DataAsset);
    }
    return Weapon;
}
```

5. Future Development Plans

This section outlines the strategic roadmap for the next phases of development, focusing on elevating the game's innovation, user experience, and technical robustness.

1. Unique Gameplay Features

- Objective: Innovate with unique gameplay mechanics to distinguish our game in the FPS genre.
- Approach: Prototype new player abilities, interactive elements, and weapon mechanics, ensuring they contribute to a unique and engaging player experience.

2. Enhanced Networking and Replication

- Objective: Deliver a seamless and fair multiplayer experience through improved network handling and replication.

- Approach: Deepen the integration with Unreal Engine's networking capabilities, focusing on optimizing data exchanges and minimizing latency for real-time player interactions.

3. Advanced Zombie AI

- Objective: Create a more immersive and challenging environment with sophisticated zombie AI that exhibits unpredictable and adaptive behaviors.
- Approach: Employ advanced AI techniques, including machine learning and state-based logic, to enhance zombie interactions and tactics, ensuring they provide a constant challenge to players.

4. Expanded Map Creation Capabilities

- Objective: Diversify the game's world with a broader range of intricate and engaging maps.
- Approach: Master advanced techniques in Unreal Engine for map creation, emphasizing strategic design, environmental storytelling, and optimization to enrich the player's exploration and combat experience.

5. Inventory System

- Objective: Implement a comprehensive and intuitive inventory system for better management of items, weapons, and equipment.
- Approach: Develop a user-friendly inventory interface supported by a robust backend, allowing for easy item management, categorization, and quick selection during gameplay.

6. Improved User Interface (UI)

- Objective: Refine the game's UI to enhance overall usability, aesthetics, and player interaction.
- Approach: Redesign the UI based on player feedback and usability testing, focusing on streamlining navigation, improving information display, and ensuring a cohesive visual style.

7. Continued Adoption of Best Development Practices

- Objective: Ensure the game's development is sustainable, scalable, and maintainable through the use of best practices.

- Approach: Maintain rigorous standards for code quality, leveraging interfaces, design patterns, and regular refactoring. Encourage a culture of continuous learning and adaptation to new and effective development methodologies.

Conclusion

By prioritizing these development areas, the game is set to evolve significantly, offering players a unique, immersive, and technically polished experience. This roadmap not only showcases a commitment to quality and innovation but also demonstrates an understanding of the importance of both player-facing features and the technical foundation that supports them.

6. Conclusion

This document has provided a detailed exploration of the cutting-edge weapon creation workflow developed for our FPS game, showcasing the technical prowess, innovative solutions, and forward-thinking design principles that define our approach to game development. From the initial automation of asset loading through Python scripting to the sophisticated use of a Proxy Weapon system and the dynamic instantiation of weapons via a Factory Method, each step of the process has been carefully crafted to enhance game performance, scalability, and player immersion.

The strategic integration of comprehensive DataAssets, encompassing not just visual and functional properties but also gameplay elements like crosshairs, bullet actors, and particles, underscores our commitment to creating a rich, immersive experience for players. Our modular system not only facilitates ease of expansion and customization but also ensures that each weapon possesses a unique identity and gameplay impact, enriching the overall game environment.

As the architect of this workflow, my goal was to push the boundaries of what is possible within Unreal Engine 5, leveraging its capabilities to their fullest while also introducing custom solutions to meet our specific development challenges. The result is a weapon system that is not only technically advanced but also deeply integrated with the game's design philosophy, offering a seamless blend of aesthetics, functionality, and gameplay dynamics.

Looking ahead, the foundations laid by this weapon creation process not only speak to the potential for future content development and game expansion but also reflect my dedication to innovation, efficiency, and quality in game development. I hope that this document serves not only as a showcase of what has

been achieved but also as an inspiration for future projects, illustrating the power of creative problem-solving and technical excellence in bringing visionary game concepts to life.

Thank you for taking the time to explore the intricacies of our game's weapon creation workflow. I look forward to the opportunity to discuss this work further and to explore how these approaches and technologies can be adapted and expanded upon in new and exciting game development projects.