Designing Custom State Machine Instead of Animation State Machine in Unity for Developing 3rd Person Action Game

Pavan P Y[1], Shanu Gour[2], Lalit Kumar P Bhaiya[3]

[1]*Student, Department of CSE, Bharti Vishwavidyalaya, Durg, Chhattisgarh, India*

[2]*Assistant Professor, Department of CSE, Bharti Vishwavidyalaya, Durg, Chhattisgarh, India*

[3]*Associate Professor, Department of CSE, Bharti Vishwavidyalaya, Durg, Chhattisgarh, India*

Corresponding Author: pavanpy444@gmail.com

Abstract— **The gaming industry has experienced unprecedented growth, evolving into a global powerhouse with diverse platforms and a vast consumer base. Market leaders such as Sony, Microsoft, and Nintendo continue to dominate the console space, while PC gaming thrives with companies like Valve and Epic Games. Mobile gaming, led by giants like Tencent and Apple, has become a major revenue driver, reaching billions of users worldwide. The rise of cloud gaming services, exemplified by Google Stadia and Microsoft's xCloud, signals a shift toward accessible, subscription-based gaming experiences. In this regard, the choice of a game development engine depends on the project's requirements, the team's expertise, and the desired platform. Unity and Unreal Engine stand out as industry leaders, each with its strengths and use cases, while other engines like Godot, CryEngine, and Lumberyard cater to specific needs within the diverse landscape of game development. Unity is a versatile and widely used game development engine known for its accessibility, cross-platform support, and flexibility. It enables developers to create 2D, 3D, augmented reality (AR), and virtual reality (VR) games. Unity uses C# as its primary scripting language and offers a robust Asset Store for pre-built assets and plugins. Design pattern implementation serves as the backbone of game development due to its pivotal role in enhancing code structure, scalability, and maintainability. By incorporating design patterns, developers can streamline the development process, leading to more efficient workflows and reduced development time. These patterns provide standardized solutions to common design problems, promoting code reuse and modularization, which are crucial for managing the complexity of game systems. Additionally, design patterns facilitate collaboration among team members by establishing a common language and framework for communication. They enable developers to create flexible and adaptable codebases that can easily accommodate changes and updates throughout the development lifecycle. Furthermore, design patterns promote best practices and coding standards, leading to cleaner, more readable code that is easier to debug and maintain. In the dynamic and fast-paced world of game development, where innovation and iteration are key, design patterns provide a solid foundation upon which developers can build immersive and engaging gaming experiences. The use of animation state machines in Unity for 3rd person action games introduces a range of challenges, encompassing synchronization issues, performance bottlenecks, responsiveness concerns, visual glitches, scalability limitations, and collaboration difficulties. Addressing these challenges is imperative to ensure the development of immersive, fluid, and engaging gaming experiences that meet the expectations of modern gamers. The proposed method aims to develop a custom state machine which will provide several benefits like Flexibility and Control, Seamless Integration with Game Logic, Responsive and Realistic Gameplay, Optimized Performance, Dynamic State Changes that are adaptable to changing conditions, Scalability and Support for Non-Animation States.**

Index Terms—**Animation State Machine, Unity Real-Time Development Platform, State Machine Design Pattern, Cross-Platform Development, Direct Script Integration.**

1. Introduction

The gaming industry has experienced unprecedented growth, evolving into a global powerhouse with diverse platforms and a vast consumer base. Market leaders such as Sony, Microsoft, and Nintendo continue to dominate the console space, while PC gaming thrives with companies like Valve and Epic Games. Mobile gaming, led by giants like Tencent and Apple, has become a major revenue driver, reaching billions of users worldwide. The rise of cloud gaming services, exemplified by Google Stadia and Microsoft's xCloud, signals a shift toward accessible, subscription-based gaming experiences. Revenue projections for the next decade suggest sustained industry expansion, with estimates surpassing $200 billion annually. Emerging technologies like virtual reality (VR) and augmented reality (AR) are poised to play a pivotal role, offering immersive and innovative gaming experiences. Esports, led by organizations like Tencent-owned Riot Games and Activision Blizzard, continues its meteoric rise, attracting massive audiences and lucrative sponsorships.

Manuscript revised July 10, 2024; accepted July 11, 2024. Date of publication July 15, 2024.
This paper available online at www.ijprse.com
ISSN (Online): 2582-7898; SJIF: 5.59

The ongoing integration of blockchain and non-fungible tokens (NFTs) into gaming ecosystems further exemplifies the industry's adaptability and willingness to embrace cutting-edge technologies. As gaming becomes increasingly intertwined with other forms of entertainment and social interaction, the industry's future seems promising, characterized by continuous innovation, diverse gaming experiences, and a flourishing global market. In this regard, the choice of a game development engine depends on the project's requirements, the team's expertise, and the desired platform. Unity and Unreal Engine stand out as industry leaders, each with its strengths and use cases, while other engines like Godot, CryEngine, and Lumberyard cater to specific needs within the diverse landscape of game development.

Unity is a versatile and widely used game development engine known for its accessibility, cross-platform support, and flexibility. It enables developers to create 2D, 3D, augmented reality (AR), and virtual reality (VR) games. Unity uses C# as its primary scripting language and offers a robust Asset Store for pre-built assets and plugins.

A. *Key Features:*

- Cross-Platform Development: Unity supports multiple platforms, including PC, consoles, mobile devices, and web browsers. This allows developers to create games for a broad range of devices and operating systems.
- User-Friendly Interface: Unity's intuitive interface and drag-and-drop functionality make it accessible for both beginners and experienced developers. The visual editor simplifies scene creation, asset management, and overall game design.
- Large Community and Documentation: Unity boasts a vast and active community, providing ample resources for learning and problem-solving. Extensive documentation, tutorials, and forums contribute to the engine's accessibility and support.
- Asset Store: The Unity Asset Store is a marketplace where developers can find and sell assets, plugins, and tools. This accelerates development by offering a wide array of pre-built resources.

Unreal Engine, developed by Epic Games, is renowned for its cutting-edge graphics, realistic physics, and high-quality visual effects. It is a powerful engine often chosen for AAA game development, architectural visualization, and immersive experiences.

B. *Key Features:*

- High-Quality Graphics: Unreal Engine is acclaimed for its stunning visuals and realistic rendering capabilities. It supports advanced graphics features like dynamic lighting, global illumination, and detailed particle effects.
- Blueprint Visual Scripting: Unreal Engine offers a visual scripting system called Blueprints, allowing developers to create gameplay mechanics and logic without extensive coding. This makes it accessible to designers and artists.
- Virtual Production: Unreal Engine gained prominence in film and TV production for its virtual production capabilities. It enables real-time rendering on set, allowing filmmakers to visualize scenes before shooting.
- Marketplace: Similar to Unity's Asset Store, Unreal Engine has the Unreal Marketplace, providing a range of assets, plugins, and tools for developers to enhance their projects.

C. *Other Game Development Engines:*

Godot Engine: Godot is an open-source game engine that supports 2D and 3D game development. It has a unique scene system, built-in visual script editor, and supports multiple scripting languages, including GDScript and C#.

CryEngine: CryEngine, known for its impressive graphics, is utilized for creating visually stunning games. It includes powerful tools for level design, terrain editing, and rendering realistic environments.

Lumberyard: Amazon Lumberyard is a game engine integrated with AWS cloud services. It offers a range of features, including visual scripting, VR development, and networking capabilities for online multiplayer games.

2. Literature Review

A. *Understanding Design Patterns*

Design patterns are recurring solutions to common problems encountered in software design. They provide a template for solving specific issues while fostering code reuse, maintainability, and scalability. Design patterns are not code snippets but rather high-level templates that guide developers in crafting effective, modular, and flexible software.

1) *Types of Design Patterns*

- Singleton Pattern: Ensures a class has only one instance and provides a global point of access to it. Useful for scenarios where a single point of control is necessary, such as managing configurations or logging.
- Factory Method Pattern: Defines an interface for creating an object but allows subclasses to alter the type of objects that will be created. Useful for designing flexible frameworks where the exact class of the created object is not known until runtime.
- Adapter Pattern: Allows incompatible interfaces to work together. It acts as a bridge, enabling the interface of a class to be used as another interface. Useful when integrating new features without modifying existing code.
- Decorator Pattern: Attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending

functionality. Useful for scenarios where the composition of behavior is more flexible than static inheritance.

- Observer Pattern: Defines a one-to-many dependency between objects, ensuring that when one object changes state, all its dependents are notified and updated automatically. Useful for implementing distributed event handling systems.
- Strategy Pattern: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the client choose the algorithm at runtime. Useful for scenarios where different algorithms need to be interchangeable.

B. *Benefits of Design Patterns*

- Code Reusability and Maintainability
- Design Clarity and Communication

Observer design pattern in detail:

The Observer Design Pattern is a behavioural pattern that defines a one-to-many dependency between objects, ensuring that when one object (the subject) changes its state, all its dependents (observers) are notified and updated automatically. This pattern is widely used to establish a loosely coupled communication mechanism between objects, promoting flexibility and maintainability in software systems.

1) *Key Components and Workflow:*

- Subject: Represents the object being observed. It maintains a list of observers and provides methods to register, remove, and notify observers of changes.
- Observer: Defines an interface with an update method that is called by the subject when its state changes. Concrete observer implementations define how they respond to updates.
- Concrete Subject: Extends the subject class and holds the actual state. It notifies observers when its state changes by calling their update methods.
- Concrete Observer: Implements the observer interface and specifies how it should respond to updates from the subject. Multiple concrete observer classes can subscribe to a single subject.

C. *State machine design pattern in detail:*

The State Machine Design Pattern is a behavioral pattern that allows an object to alter its behavior when its internal state changes. This pattern is particularly useful when an object's behavior is dependent on its internal state, and the transitions between these states are well-defined. The primary goal is to encapsulate the behavior associated with each state and allow the object to transition seamlessly between states based on certain conditions.

1) *Key Components*

- State: Represents a distinct state of the object. Each state encapsulates a specific set of behaviors associated with the object when it is in that state.
- Context: Maintains a reference to the current state and provides an interface for clients to interact with the object. The context delegates state-specific behavior to the current state object.
- Concrete States: Implement specific behaviors associated with a particular state. Each concrete state provides its own implementation of the methods defined by the state interface.

D. *Implementation Steps*

1. Define States
2. Create State Interface
3. Implement Concrete States
4. Create Context Class
5. Client Interaction

1) *State Machine Basics in Unity*

The basic idea is that a character is engaged in some particular kind of action at any given time. The actions available will depend on the type of gameplay but typical actions include things like idling, walking, running, jumping, etc. These actions are referred to as states, in the sense that the character is in a "state" where it is walking, idling or whatever. In general, the character will have restrictions on the next state it can go to rather than being able to switch immediately from any state to any other.

For example, a running jump can only be taken when the character is already running and not when it is at a standstill, so it should never switch straight from the idle state to the running jump state. The options for the next state that a character can enter from its current state are referred to as state transitions. Taken together, the set of states, the set of transitions and the variable to remember the current state form a state machine.

The states and transitions of a state machine can be represented using a graph diagram, where the nodes represent the states and the arcs (arrows between nodes) represent the transitions. You can think of the current state as being a marker or highlight that is placed on one of the nodes and can then only jump to another node along one of the arrows.

The importance of state machines for animation is that they can be designed and updated quite easily with relatively little coding. Each state has a Motion associated with it that will play whenever the machine is in that state. This enables an animator or designer to define the possible sequences of character actions and animations without being concerned about how the code will work.

3. Problem Identification

A. *Game Design Challenges*

Designing and implementing AAA games for PC and consoles involves overcoming a myriad of technical, coding, and architectural challenges. These challenges span various aspects of game development, from graphics and physics to networking and optimization. Here's a detailed breakdown of these challenges:

1) *Graphics and Rendering:*
   - Realistic Graphics: Achieving realistic graphics demands advanced rendering techniques, such as physically-based rendering (PBR) and global illumination, which can strain hardware capabilities.
   - Optimization: Balancing visual fidelity with performance on a variety of hardware configurations requires extensive optimization, considering factors like texture streaming, level-of-detail (LOD), and efficient shader programming.
2) *Physics and Simulation:*
   - Realistic Physics: Simulating realistic physics for character movements, object interactions, and environmental effects poses a challenge, demanding complex algorithms and precise optimizations.
   - Collision Detection: Implementing efficient collision detection algorithms is crucial for handling complex environments and interactions between game entities.
3) *AI and Pathfinding:*
   - Intelligent NPCs: Developing advanced AI for non-player characters (NPCs) with realistic behaviours, decision-making, and adaptive learning presents a significant challenge.
   - Dynamic Pathfinding: Creating dynamic and efficient pathfinding algorithms that adapt to changing environments and handle complex terrain is crucial for immersive gameplay.
4) *Networking:*
   - Multiplayer Challenges: Implementing robust and low-latency networking code for seamless multiplayer experiences requires addressing issues like synchronization, lag compensation, and anti-cheat mechanisms.
   - Scalability: Designing network architecture that scales for a large number of players while maintaining a stable connection introduces challenges related to server load balancing and data synchronization.
5) *Optimization:*
   - Hardware Diversity: Ensuring optimal performance across a wide range of PC and console hardware configurations requires thorough optimization, including parallelization, multithreading, and platform-specific optimizations.
   - Memory Management: Effectively managing memory resources to prevent bottlenecks and crashes, especially in open-world games with extensive assets, is a constant challenge.
6) *Content Creation and Integration:*
   - Asset Pipelines: Developing efficient asset pipelines for handling massive amounts of audio, visual, and gameplay assets, while maintaining version control and collaboration among large development teams, is crucial.
   - Cross-Disciplinary Collaboration: Facilitating seamless collaboration between artists, designers, and programmers to integrate diverse assets and ensure consistency in the game world poses organizational challenges.
7) *Security:*
   - Piracy and Cheating: Implementing robust security measures to prevent piracy and cheating in online multiplayer games requires continuous updates and monitoring.
8) *Platform-specific Challenges:*
   - Console Optimization: Optimizing games for specific console architectures and ensuring compliance with console certification requirements adds an extra layer of complexity.
   - PC Hardware Variability: Dealing with the diverse range of PC hardware configurations and ensuring compatibility across different operating systems can be challenging.

B. *Challenges of applying design patterns*

Designing AAA games involves numerous challenges related to the application of design patterns. These challenges arise due to the complex and dynamic nature of game development, where performance, scalability, and maintainability is critical. Here's a summary of the challenges faced in the application of design patterns during the design of AAA games:

- Performance Optimization with Overhead Concerns: While design patterns provide elegant solutions to common problems, they can introduce overhead. Striking a balance between clean design and optimal performance is crucial, especially in resource-intensive AAA games.
- Scalability with Adaptability to Game Size: Design patterns should scale seamlessly as the game project grows in size and complexity. Ensuring that patterns remain effective and maintainable across large codebases and expansive game worlds is a constant challenge.
- Real-time Constraints with Game Loop Integration: Integrating design patterns into the real-time game loop requires careful consideration. Patterns must be efficient and not compromise the responsiveness of the game, particularly in fast-paced, action-packed scenarios.
- Concurrency and Multithreading with Parallel Processing: Designing patterns that can effectively harness the power of parallel processing and multithreading is crucial for optimizing game performance, especially in rendering, physics, and AI computations.
- Resource Management with Memory Efficiency: Design patterns should address memory management challenges to prevent memory leaks and optimize resource utilization. This is particularly critical in

AAA games with vast amounts of assets and dynamic environments.

- Cross-disciplinary Collaboration with Communication Between Teams: Ensuring effective communication and collaboration between different development teams, including artists, designers, and programmers, is essential. Design patterns need to facilitate collaboration and integration of diverse elements seamlessly.
- Adaptability to Game Genres with Genre-specific Challenges: Different game genres may have unique requirements. Design patterns must be adaptable to these diverse needs, whether it's handling complex AI behaviours, intricate physics simulations, or dynamic storytelling mechanics.
- Maintainability with Long-term Viability: Design patterns should contribute to the maintainability of the codebase over the game's lifecycle. Preventing code bloat, ensuring ease of debugging, and facilitating updates are ongoing challenges.
- User Interface (UI) Design with Dynamic UI Requirements: Design patterns for UI must accommodate the dynamic nature of AAA game interfaces. Patterns need to handle various screen resolutions, input devices, and interactive elements while maintaining a smooth and immersive user experience.
- Emergent Gameplay with Unpredictable Scenarios: Design patterns may face challenges when dealing with emergent gameplay, where unexpected interactions between game elements create unique scenarios. Patterns need to be flexible enough to handle these unforeseen situations.
- Data-driven Design with Flexible Data Structures: AAA games often rely on data-driven design for flexibility. Design patterns should facilitate the integration of dynamic data structures, allowing designers to tweak game parameters without extensive code changes.
- Cross-platform Considerations with Platform-specific Patterns: Designing patterns that are platform-agnostic or easily adaptable to different gaming platforms (PC, consoles, etc.) is crucial for achieving broad market accessibility.

C. *Limitations of using Unity Animation State Machines*

The implementation of animation state machines in Unity for the development of 3rd person action games presents a myriad of challenges that can significantly impact the overall gaming experience. One major concern revolves around the intricate synchronization of character animations, where transitioning seamlessly between action states often results in disjointed and unrealistic movements. This issue not only compromises the visual aesthetics of the game but can also hinder player immersion.

Furthermore, the complexity of managing multiple animation states can lead to programming inefficiencies, causing performance bottlenecks and impacting the game's responsiveness. Balancing the responsiveness of character controls with the intricacies of combat animations poses a significant technical challenge, often resulting in delayed or inaccurate player inputs during critical gameplay moments.

The struggle to achieve a cohesive and visually appealing animation blend becomes particularly pronounced when integrating diverse character movements, such as jumping, climbing, and melee attacks. The potential for unintended animation interruptions or glitches during dynamic sequences creates a risk of frustrating player experiences and negatively affecting the game's overall polish.

Additionally, the scalability of animation state machines in larger game projects can become a stumbling block, making it difficult for developers to maintain and expand the system as the game evolves. This lack of scalability can impede the addition of new features, characters, or animations, limiting the game's potential for growth and innovation.

Collaboration between animators and programmers becomes challenging due to the inherent complexity of animation state machines, often resulting in miscommunications and delays in implementing desired changes or improvements. This discord can hinder the creative workflow and compromise the timely delivery of a polished gaming experience.

In summary, the use of animation state machines in Unity for 3rd person action games introduces a range of challenges, encompassing synchronization issues, performance bottlenecks, responsiveness concerns, visual glitches, scalability limitations, and collaboration difficulties. Addressing these challenges is imperative to ensure the development of immersive, fluid, and engaging gaming experiences that meet the expectations of modern gamers.

4. Methodology

A. *Software*

Developing games in Unity with C# requires a specific set of system and software requirements.

1) *Software Requirements:*

- Unity Hub: Unity Hub is a management tool for Unity projects. It allows you to install and manage different versions of Unity.
- Unity Editor: Unity Editor is the core development environment where you design, build, and test your games.
- Visual Studio or Visual Studio Code: Unity uses Visual Studio as the default integrated development environment (IDE). Visual Studio Code is also a popular choice with additional plugins for Unity.

2) *Plugins and Frameworks:*

- TextMeshPro: Unity's TextMeshPro is a powerful text rendering tool that provides enhanced text and font capabilities. It is widely used for creating dynamic and stylized text in games.

- Cinemachine: Cinemachine is a camera system for Unity that provides dynamic and procedural camera movements. It's useful for creating cinematic and visually appealing gameplay experiences.
- Post-Processing Stack: The Post-Processing Stack in Unity enhances visual effects in games. It includes features like ambient occlusion, bloom, and color grading to improve the overall look of your game.

3) Additional Recommendations:

- UI/UX Design Tools: Depending on your game's requirements, you might use design tools like Adobe XD, Sketch, or Figma for UI/UX design.
- 3D Modelling Software (Optional): For creating 3D models, you may use software like Blender, Autodesk Maya, or Cinema 4D.

B. Implementation

Unity, one of the most popular game development engines, relies heavily on C# as its primary scripting language. Coding in C# within the Unity environment offers a powerful and flexible approach to game development.

1) Player

i. Creating the State Machine for Player:

Define the States: Create an enumeration to represent all possible states of the player.

State Machine Base Class: Create a base class for the state machine that will handle the current state and transitions.

Player Controller: Implement the player controller to manage states and handle state transitions.

ii. Implementing Specific States

Each state will inherit from the base State class and override necessary methods.

Idle State
Attack State
Block State

iii. Implementing Complex States

Dodge State
Impact State
Dead State

iv. Handling Movement States

Fall Hang State
Jump State
Pull Up State

2) Enemy

i. Creating the State Machine for Enemy:

Base State
Idle State
Chase State
Attack State
Impact State
Dead State

ii. Implementing Enemy AI:

Enemy Controller
Pathfinding
Behavior Tree

3) Unity Specifics

i. Cinemachine

Cinemachine is a powerful and flexible camera system for Unity 3D that simplifies the process of creating dynamic, high-quality camera behaviors for games and interactive applications. Key Features of Cinemachine:

- Smart Camera Controls: Cinemachine offers smart camera controls that automatically adjust to provide the best view of the scene or action. It handles camera transitions, composition, and follows targets smoothly.
- Virtual Cameras: Instead of manipulating a single camera, Cinemachine uses virtual cameras that define different camera behaviors and settings. Developers can switch between these virtual cameras seamlessly to create diverse cinematic effects.
- Advanced Camera Shake: Cinemachine provides robust camera shake effects, enhancing the realism and impact of actions like explosions, impacts, and rapid movements.
- Blend and Cut: Cinemachine can blend smoothly between different camera states or cut instantly, giving developers control over the pacing and style of camera transitions.
- Extensions and Customization: Cinemachine includes numerous extensions for custom behaviors, such as collision detection, damping, and look-ahead, allowing for extensive customization and fine-tuning.
- Integration with Timeline: Cinemachine integrates seamlessly with Unity’s Timeline tool, enabling developers to choreograph complex camera sequences alongside animations and events.

ii. Action Map

Creating an action map in Unity 3D involves setting up input controls for your game using the Input System package. The Input System offers a more flexible and robust way to handle player inputs than the legacy input manager.

Step 1: Installing the Input System Package

First, you need to install the Input System package. Open the Package Manager (Window > Package Manager), search for "Input System," and install it. After installation, Unity will prompt you to restart the editor to enable the new input system.

Step 2: Setting Up the Input Actions

1. Create Input Actions Asset:

• Go to the Project window, right-click, and select Create > Input Actions. Name this asset "PlayerInputActions".

2. Open the Input Actions Editor:

• Double-click on the "PlayerInputActions" asset to open the Input Actions editor.

3. Creating an Action Map:

• Click the + button to add a new action map. Name it "Player".

4. Adding Actions:

• Within the "Player" action map, add actions for different player inputs. For example, add actions named "Move", "Jump", "Attack", and "Dodge".

5. Defining Bindings:

• For each action, define the control bindings. For "Move", add a 2D Vector Composite and bind it to the "WASD" keys on the keyboard and the left stick on the gamepad.

• For "Jump", bind it to the spacebar on the keyboard and the "A" button on the gamepad.

• For "Attack", bind it to the left mouse button on the keyboard and the "X" button on the gamepad.

• For "Dodge", bind it to the left shift key on the keyboard and the "B" button on the gamepad.

Step 3: Generating the C# Class

1. Generate C# Class:

• Click the Generate C# Class button in the Input Actions editor. Name the class "PlayerInputActions" and click Apply.

Step 4: Integrating with Player Script

1. Create a Player Script:

• Create a new C# script named "PlayerController" and attach it to the player GameObject.

2. Referencing the Input Actions:

• In the "PlayerController" script, create a reference to the "PlayerInputActions" class and implement methods to handle the actions.

iii. Character Movement Physics

Start by creating a new GameObject in your Unity scene to represent the character. This object will serve as the root of your character's hierarchy.

To enable physics-based movement, add the following components to the "Player" GameObject:

1. Rigidbody:

• Select the "Player" GameObject, click Add Component, and choose Rigidbody. This component makes the GameObject subject to physics simulations.

2. Collider:

• Add a Capsule Collider to match the shape of the character. This collider will handle collisions with the environment.

3. Script:

• Create a new C# script named "PlayerController" and attach it to the "Player" GameObject. This script will manage the character's movement logic.

To make the character face the direction of movement, you can update the character's rotation in the Move method.

Raycasting for Ground Detection to detect the ground provides more accurate ground checks and helps handle slopes and uneven terrain.

Handling Slopes adjusting the movement direction based on the surface normal.

Physics Settings

Adjust Unity's physics settings for optimal performance.

1. Fixed Timestep:

• Set an appropriate fixed timestep in Edit > Project Settings > Time.

2. Collision Detection:

• Choose between discrete and continuous collision detection based on your needs.

Efficient Raycasting: Minimize performance impact by optimizing raycasts. Only perform necessary checks and avoid redundant raycasts.

Object Pooling: For effects like particle systems or instantiated objects, use object pooling to reduce garbage collection and improve performance.

5. Results

System Used to perform simulation:

- Operating System: Windows 11 23H2
- 13th Gen Intel® Core™ i9-13980HX 2.2 GHz (24 cores: 8 P-cores and 16 E-cores)
- Graphics Card: NVIDIA® GeForce RTX™ 4070 Laptop GPU
- Memory (RAM): 16 GB RAM
- Storage: 1TB SSD

The performance of AAA (Triple-A) games is measured through various metrics and benchmarks to ensure smooth and enjoyable gameplay experiences. Some criteria that were considered:

- Frame Rate (FPS): Frame rate measures the number of frames rendered per second. Higher frame rates, typically 30 FPS or above, contribute to smoother animations and more responsive controls. Many AAA games aim for 60 FPS or even higher for a more immersive experience.
- Resolution: The resolution of the game refers to the number of pixels displayed on the screen. Higher resolutions, such as 1080p (Full HD), 1440p (Quad HD), or 4K, result in sharper and more detailed visuals. The choice of resolution can impact performance, and optimizing games for various resolutions is crucial for a broad player base.
- Graphics Settings: AAA games often provide a range of graphics settings that users can adjust based on their hardware capabilities. These settings include options for texture quality, shadow quality, anti-aliasing, and other visual effects.
- Load Times: Load times are critical for a seamless gaming experience. Faster load times contribute to a smoother flow between levels or scenes. The performance is evaluated based on how quickly the

game loads assets, textures, and levels.

- Stability and Consistency: Stability refers to the reliability of the game's performance over time. Consistency in frame rate and responsiveness is crucial to avoid issues like stuttering, freezing, or sudden drops in performance during gameplay.
- CPU and GPU Utilization: Monitoring the utilization of the central processing unit (CPU) and graphics processing unit (GPU) provides insights into how efficiently the game utilizes hardware resources. Balanced utilization ensures optimal performance without bottlenecks.

Table.1.
Tabulated Results Comparison

Criteria	Animation State Machine	Custom State Machine	Custom State Machine Scaled with load
Frame Rate (FPS)	180 to 300 FPS	240 to 360 FPS	240 to 360 FPS
CPU Milliseconds (ms)	Between 4ms to 6ms	Below 4.5ms	Below 4.5ms
Resolution	4K	4K	4K
Stability	Rare screen freezes	No crashes/No screen freezing	No crashes/No screen freezing
GPU Usage in %	Up to 85%	Up to 85%	Up to 85%
CPU Usage in %	Up to 16%	Up to 20%	Up to 20%
Memory	Up to 8GB	Up to 8GB	Up to 8GB

6. Conclusion & Future Scope

A. Conclusion:

Using custom state machines in Unity for game development provides several benefits, offering greater control, flexibility, and efficiency in managing the logic and behavior of game entities. Below are the details:

- Flexibility and Control: Tailored to Game Requirements: Custom state machines allow developers to design and implement states that precisely fit the specific requirements of the game. This flexibility is particularly valuable in scenarios where Unity's built-in Animator Controller might not provide the required level of customization.
- Seamless Integration with Game Logic: Direct Script Integration: Custom state machines are often implemented directly in scripts using C#. This tight integration allows for seamless coordination between game logic, user input, and state transitions.
- Responsive and Realistic Gameplay: Fine-tuned Transitions: With a custom state machine, developers can fine-tune state transitions to create responsive and realistic gameplay. This is crucial for character movements, combat animations, and other dynamic behaviors, providing a more immersive experience for players.
- Optimized Performance: Reduced Overhead: Custom state machines can be optimized for performance, reducing unnecessary overhead associated with generic solutions. This optimization is especially important in resource-intensive games where performance is a critical factor.
- Dynamic State Changes: Adaptable to Changing Conditions: Custom state machines are adaptable to changing game conditions. They can dynamically adjust states based on variables such as health, environmental factors, or player progression, allowing for a more dynamic and engaging gaming experience.
- Support for Non-Animation States: Beyond Animation Control: While Unity's Animator Controller is primarily focused on animation, custom state machines can handle a broader range of states. This includes non-animation states related to gameplay mechanics, AI behavior, or any other aspect of the game that requires state-based control.

B. Future Scope:

Implementing a custom state machine in Unity 3D for developing a third-person combat game opens up numerous future possibilities. A well-designed state machine enhances modularity, maintainability, and scalability of the game's codebase, facilitating easier updates and feature additions. Future enhancements could include more sophisticated AI behaviors, allowing enemies and NPCs to exhibit more realistic and varied responses in combat scenarios.

Additionally, integrating more advanced animation blending techniques and transitions can lead to smoother and more natural character movements. As the game evolves, the state machine can accommodate new combat mechanics, such as advanced combos, special attacks, and defensive maneuverers, without significant refactoring.

Expanding multiplayer capabilities becomes more feasible with a robust state machine, as managing multiple player states and synchronizing them over a network can be handled more systematically. Moreover, the state machine can be extended to support various game modes, each with unique rules and player interactions, enhancing replay ability and user engagement.

Finally, the modular nature of a custom state machine allows for easier porting to different platforms, ensuring the game can reach a wider audience across PCs, consoles, and mobile devices. This foundational work sets the stage for continuous improvement and expansion, ensuring the game's longevity and success.

References

[1]. Wahyu Safitra, Ahmad Faisol, Suryo Adi Wibowo, “Application of the Finite State Machine Method to Non-Player Character (NPC) Action Strategy Game 'Ouroboros'”, Jurnal Mahasiswa Teknik Informatika, 04 September 2023,

[2]. Muhammad Khafidh Aulia, Ali Mahmudi, Sentot Achmadi, “Application of the finite state machine method in android-based pandemic nightmare game”, Jurnal Mahasiswa Teknik Informatika, 07 September 2023.

[3]. Devang Jagdale, “Finite State Machine in Game Development”, International Journal of Advanced Research in Science, Communication and Technology, 08 September 2023.

[4]. Enggar Adji Laksono, “Mathematics Education Game Using the Finite State Machine Method to Implement Virtual Reality in Game Platformer”, Inform: Jurnal Ilmiah Bidang Teknologi Informasi dan Komunikasi, 09 September 2023.

[5]. Robert Collier, “A Computer Game to Teach Finite-State Machine Artificial Intelligence to First-Year Undergraduates”, IEEE Xplore, 17 September 2023.

[6]. Jiacun Wang, William Tepfenhart, “Formal Methods in Computer Science – Finite State Machine”, Taylor & Francis Group, 28 September 2023.

[7]. Jeff W. Murray, “C# Game Programming Cookbook for Unity 3D”, Taylor & Francis Group, 28 September 2023.

[8]. Alex Okita, “Learning C# Programming with Unity 3D, second edition”, Taylor & Francis Group, 28 September 2023.

[9]. Juan Wu, “Research on roaming and interaction in VR game based on Unity 3D”, IEEE Xplore, 17 September 2023.