Implementing Custom State Machine for 3rd Person Battleground in Unity

Pavan P Y[1], Shanu Gour[2]

[1]MTech Scholar, Department of CSE, Bharti Vishwavidyalaya, Durg, C.G., India
[2]Assistant Professor, Department of CSE, Bharti Vishwavidyalaya, Durg, C.G., India

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.

Keywords: 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. 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.

Key Features:

1. **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.
2. **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.
3. **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.
4. **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.

2. Limitation 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.

3. Methodology

Software

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

Software Requirements:

1. Unity Hub: Unity Hub is a management tool for Unity projects. It allows you to install and manage different versions of Unity.
2. Unity Editor: Unity Editor is the core development environment where you design, build, and test your games.
3. 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.

Plugins and Frameworks:

1. 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.
2. Cinemachine: Cinemachine is a camera system for Unity that provides dynamic and procedural cam-

era movements. It's useful for creating cinematic and visually appealing gameplay experiences.
3. 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.

Additional Recommendations:

1. 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.
2. 3D Modelling Software (Optional): For creating 3D models, you may use software like Blender, Autodesk Maya, or Cinema 4D.

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.

A. 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

B. 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

C. 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:

1. 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.
3. 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.
4. Advanced Camera Shake: Cinemachine provides robust camera shake effects, enhancing the realism and impact of actions like explosions, impacts, and rapid movements.
5. 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.
6. 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.
7. 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.

4. Result & Discussions

System Used to perform simulation:

1. Operating System: Windows 11 23H2
2. 13th Gen Intel® Core™ i9-13980HX 2.2 GHz (24 cores: 8 P-cores and 16 E-cores)
3. Graphics Card: NVIDIA® GeForce RTX™ 4070 Laptop GPU
2. Memory (RAM): 16 GB RAM

3. 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:

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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

5. Conclusion & Future Scope

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:

1. ***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.
2. ***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.
3. ***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.
4. ***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.

2. ***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.
3. ***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.

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 manoeuvrers, 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.

6. 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, https://ejournal.itn.ac.id/index.php/jati/article/view/2828
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, https://ejournal.itn.ac.id/index.php/jati/article/view/3221
3. Devang Jagdale, "*Finite State Machine in Game Development*", International Journal of Advanced Research in Science, Communication and Technology, 08 September 2023, https://ijarsct.co.in/Paper2062.pdf
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, https://ejournal.unitomo.ac.id/index.php/inform/article/view/1860
5. Robert Collier, "*A Computer Game to Teach Finite-State Machine Artificial Intelligence to First-Year Undergraduates*", IEEE Xplore, 17 September 2023, https://ieeexplore.ieee.org/document/9569277
6. Jiacun Wang, William Tepfenhart, "*Formal Methods in Computer Science – Finite State Machine*", Taylor & Francis Group, 28 September 2023, https://www.taylorfrancis.com/chapters/mono/10.1201/9780429184185-2/finite-state-machine-jiacun-wang-william-tepfenhart
7. Jeff W. Murray, "C# Game Programming Cookbook for Unity 3D", Taylor & Francis Group, 28 September 2023, https://www.taylorfrancis.com/books/mono/10.1201/9780429317132/game-programming-cookbook-unity-3d-jeff-murray
8. Alex Okita, "Learning C# Programming with Unity 3D, second edition", Taylor & Francis Group, 28 September 2023, https://www.taylorfrancis.com/books/mono/10.1201/9780429810251/learning-programming-unity-3d-second-edition-alex-okita
9. Juan Wu, "Research on roaming and interaction in VR game based on Unity 3D", IEEE Xplore, 17 September 2023, https://ieeexplore.ieee.org/document/9270519