Spooky House

Jamie Wales

March 5, 2025

1 Overview

1.1 Core Concept

"Spooky House" is a high-octane first-person shooter that combines the wave-based survival mechanics of Call of Duty Zombies, the fast-paced arena combat of Quake/Doom, and the bullet-time manipulation from The Matrix (PS2). Players battle against increasingly difficult waves of enemies while utilising superhuman reflexes to slow down time, perform impossible stunts, and eliminate threats with stylish precision.

1.2 Genre

"Spooky House" is first and foremost a first-person shooter. However, it combines the following subgenres to create a unique and engaging experience:

- First-Person Shooter
- Survival Horror
- Bullet-Time Action
- Arena Combat

1.3 Insperation

Here is an overview of games that has inspired "Spooky House"'s core mechanics and design

1.3.1 Call of Duty Zombies

Call of Duty Zombies is a wave-based survival mode that has become a staple of the franchise. Players must survive against increasingly difficult waves of enemies while utilising the environment and special weapons to stay alive. Spooky House takes inspiration from this mode by incorporating wave-based survival mechanics. As the game progresses the player will have access to increasingly powerful weapons and abilities to help them survive. The game will not implement the room-based system of Call of Duty Zombies, instead opting for a more open-world approach. Weapons will drop off of enemies, making it important to kill as many enemies as possible to get the best weapons.

1.3.2 Doom/Quake

Doom and Quake are fast-paced arena shooters that focus on movement and precision shooting. Players must navigate complex levels while battling against hordes of enemies. The movement system will directly integrate with score and the bullet time mechanic. Players will receive more points for killing enemies in stylish ways, such as headshots, while in bullet time or whilst moving. The game will also feature a combo system that rewards players for chaining together kills and stunts. The player's bullet time meter will increase with more stylish kills and movement, encouraging players to play aggressively and take risks, and giving them a satisfying core loop of killing enemies, gaining points, and using those points to gain more bullet time.

1.3.3 The Matrix (PS2)

The Matrix video game for PS2 featured a bullet time mechanic that allowed players to slow down time and perform impossible stunts. This mechanic will be a core feature of Spooky House, allowing players to slow down time to dodge bullets, perform acrobatic moves, and eliminate enemies with precision. The bullet time mechanic will be tied to a meter that depletes as the player uses it, encouraging strategic use of the ability. Spooky house will also feature a "focus" mechanic that allows players to slow down time even further for a short period, allowing them to perform even more impressive stunts and kills. The game will encourage use of this feature, allowing players to turn on and off bullet time at will, and rewarding them for using it in creative ways.

1.3.4 Devil May Cry

Devil May Cry is a action game that rewards players for performing stylish combos and kills. Spooky house will implement a style based score system after each wave encouraging reliability and mastery of the game. The game will also feature a variety of weapons and abilities that players can use to customise their play style and create unique combos. The game will also feature a variety of enemies with different weaknesses and attack patterns, encouraging players to adapt their playstyle and strategy to overcome each wave.

1.4 Target Audience

- FPS enthusiasts aged 18-35 who enjoy fast-paced shooters with unique mechanics
- Fans of classic gaming, particularly those with nostalgia for PlayStation 2 era aesthetics
- Action game players looking for stylish combat with high skill expression

1.5 Unique Selling Points

- Bullet Time: Slow down time while maintaining movement speed, allowing for spectacular dodges, precision aiming, and cinematic moments
- Wave Escalation: Survive increasingly difficult waves of enemies with intermittent boss encounters
- Movement Mechanics: Utilise double jumps, wall runs, and slides that become even more powerful during bullet time
- Kill Combos: Chain kills to build momentum, unlock special abilities, and earn score multipliers

2 Work complete

A lot of the current work has gone into systems which the game can built built on top of. A great deal of effort has gone into using scriptable objects to hold the data in which the systems rely. This way, the game can be easily expanded upon and changed without having to change the code. This is a great way to work as it allows for a lot of flexibility and creativity in the development process. This section will explain what features have been built thus far.

2.1 Player Movement

This is a cornerstone of the Spooky House experience, serving as the foundation upon which all other gameplay elements are built. Drawing inspiration from arena shooters like Quake and Doom, the movement system emphasizes speed, fluidity, and acrobatic potential.

2.1.1 Player Animation States

The player character utilizes a state-based animation system with smooth transitions between key states. Figure 3 shows the primary animation states and their transitions. The animation system uses boolean parameters and triggers in the Unity Animator to control transitions between states. The core animation states are:

- **Idle**: Default state when no input is detected
- **Run**: Active when movement input is detected
- **Shoot**: Triggered by left mouse button
- **Reload**: Activated by pressing R key
- **Jump/Dive**: Special movement states with jump triggered by space bar, and dive by pressing space again during the jump window

The dive roll mechanic provides both gameplay advantages (brief invulnerability) and scoring benefits (increased style multiplier).

2.1.2 Core Movement Implementation

The current implementation features a robust character controller that provides dynamic movement options (dive roll algorithm detailed in Appendix A.4):

- Variable movement speeds (walking and sprinting) with smooth transitions
- Precision jumping mechanics with adjustable force parameters
- Advanced double-jump capabilities with a timing window
- Signature dive roll maneuver for evasive action and stylish combat
- Custom ground detection using raycasting for improved responsiveness
- Full animation integration with state transitions

The dive roll mechanic is particularly significant as it represents a key element of the movement system. When a player jumps and then presses jump again within the double-jump time window, they execute a forward dive that can be used to dodge incoming fire, close distance to enemies, or position for spectacular kills that earn style points.

2.1.3 Technical Challenges Overcome

Developing this movement system presented several technical challenges that have been successfully addressed:

- Creating reliable ground detection that works with complex geometry
- Implementing smooth transitions between movement states
- Ensuring dive roll momentum and direction feel natural and responsive
- Balancing movement speeds and jump forces for satisfying gameplay

The current system uses a combination of character controller methods and custom physics calculations to achieve smooth, responsive movement that maintains player control even during complex maneuvers. The implementation carefully monitors the player's grounded state, handles input for different movement types, and manages the specialized dive roll functionality. The dive roll implementation required particular attention to detail, as it needed to determine appropriate direction based on either the player's current velocity or input direction. This ensures that players can execute stylish dives in any direction they choose, adding both tactical options and visual flair to combat scenarios.

2.2 Enemy System

The enemy system in "Spooky House" forms the cornerstone of the combat experience, providing dynamic and responsive opponents that challenge the player's skills. The implementation uses a robust state machine architecture that allows enemies to react intelligently to the player's actions.

2.2.1 State-Based AI

Enemies operate on a sophisticated state machine that transitions between six distinct behavioral states (see Appendix A.1 for the full algorithm implementation):

- **Idle**: The default state where enemies remain stationary while scanning for the player
- **Patrol**: Enemies navigate randomly through the environment using Unity's NavMesh system
- **Chase**: Upon detecting the player, enemies actively pursue using pathfinding
- **Attack**: Within range, enemies execute attacks with appropriate animations and effects
- **Hurt**: A transitional state triggered when enemies take damage
- **Dead**: Handles death animations, particle effects, and eventual removal from the scene

This state-based approach allows enemies to make contextual decisions based on player proximity, previous actions, and environmental factors. Each state has its own specific behaviors and transition conditions, creating emergent gameplay scenarios as enemies respond dynamically to the player's tactics.

2.2.2 NavMesh Integration

The enemy movement system leverages Unity's NavMesh for intelligent pathfinding, allowing enemies to navigate complex environments while pursuing the player. Key features include:

- Dynamic path calculation to track moving targets
- Stuck detection and recovery mechanisms
- Speed variations between patrol and chase states
- Variable stopping distances based on attack ranges

This integration ensures enemies can navigate realistically around obstacles while maintaining pursuit pressure on the player, regardless of the arena's complexity.

2.2.3 Scriptable Object Architecture

Enemy behaviors and attributes are defined through a flexible scriptable object system (EnemyData) that decouples data from implementation. This approach allows for:

- Easy creation of diverse enemy types without code changes
- Centralized balancing of health, damage, and speed values
- Assignment of custom animations, sounds, and visual effects
- Runtime variation of enemy capabilities

This data-driven design enables rapid iteration and easy expansion of the enemy roster, supporting the game's wave-based progression system.

2.2.4 Combat Feedback Systems

Enemies provide rich feedback during combat encounters through:

- Visual hit reactions and knockback effects
- Contextual audio cues for different states
- Particle effects for hits and death sequences
- Animation transitions that reflect current behavior

These feedback mechanisms communicate the game state clearly to the player while enhancing the satisfaction of combat encounters.

2.2.5 Technical Resilience

The enemy system includes numerous safeguards to ensure stable operation even under edge cases:

- Component verification with automatic addition of missing requirements
- NavMesh position validation to prevent navigation errors
- Error-resistant animation handling with exception management
- Graceful destruction sequence to prevent runtime errors

2.3 Wave Spawning System

The Wave Spawning System in "Spooky House" drives the game's core progression mechanics, creating an escalating challenge through procedurally generated enemy waves. This system implements the survival foundation inspired by Call of Duty Zombies while adapting it to the more open, arena-based environment.

2.3.1 Dynamic Wave Generation

The spawning system creates waves with increasing difficulty through procedural generation (algorithm provided in Appendix A.3):

- Configurable enemy count ranges that can scale with wave progression
- Randomized spawn locations within a defined radius around spawn points
- NavMesh validation to ensure enemies spawn on valid traversable surfaces
- Timed intervals between enemy spawns to create manageable combat pacing
- Cooldown periods between waves to allow players recovery time

This approach creates unpredictable but fair combat scenarios, ensuring each playthrough feels fresh while maintaining a consistent difficulty curve.

2.3.2 Enemy Type Selection

The system leverages the scriptable object architecture to create diverse enemy compositions:

- Waves draw from a configurable pool of enemy types
- Random selection ensures varied combat encounters
- Enemy data is passed to spawned instances, controlling their behavior and capabilities
- Each enemy is named contextually based on wave number and remaining count for debugging

This implementation allows for easy expansion of the enemy roster and supports natural difficulty progression as more challenging enemy types can be introduced in later waves.

2.3.3 Wave Management

The system maintains complete awareness of the current game state through:

- Active tracking of all spawned enemies
- Automatic cleanup of destroyed enemies
- Wave completion detection when all enemies are defeated
- Wave progression counters for scaling difficulty

This creates a rhythm of combat intensity and brief respite that forms the core gameplay loop of "Spooky House," pushing players to survive increasingly challenging scenarios while providing moments to recover between waves.

2.3.4 Technical Implementation

The spawning system integrates with other game systems through:

- Direct configuration of spawned enemy controllers
- Player reference passing to ensure enemies can locate their target
- Unity's NavMesh system for spawn position validation
- Coroutine-based spawning for controlled timing

The system uses a combination of Update cycle checks and coroutines to manage wave timing, enemy spawning, and cleanup operations efficiently. This hybrid approach ensures responsive wave management without creating performance bottlenecks during intense combat.

2.4 Weapon System

The weapon system in "Spooky House" provides players with diverse and satisfying combat options. Built on scriptable objects for maximum flexibility, the system supports a variety of weapon types that can be collected and cycled through during gameplay.

2.4.1 Weapon Management

The core weapon management features include:

- Automatic weapon attachment to the player character's hand bone
- Cycling between multiple weapon types during gameplay
- Configurable positioning, rotation, and scale for each weapon
- Weapon-specific parameters including damage, fire rate, and ammunition capacity
- Animation integration with the player character's animation controller

This flexibility allows for a wide range of weapon types that each feel distinct while maintaining a coherent attachment to the player model. The system uses Unity's humanoid animation rigging to automatically locate and attach weapons to the player's hand.

2.4.2 Visual Feedback

Combat feedback is critical to creating satisfying gunplay, and the weapon system provides rich visual cues:

- Muzzle flash effects using Unity's Visual Effect Graph system
- Bullet trail renderers with object pooling for performance
- Dynamic movement of trails along calculated bullet paths
- Custom crosshair expansion to indicate weapon firing
- Animation triggers for player shooting and reloading actions

Object pooling for bullet trails ensures that even during high-intensity combat with multiple rapid-fire weapons, performance remains smooth with minimal garbage collection overhead.

2.4.3 Weapon Data Architecture

The scriptable object-based weapon data system allows for:

- Easy creation and balancing of new weapon types
- Centralized management of weapon properties
- Runtime ammunition tracking
- Weapon-specific audio and visual effects

This data-driven approach streamlines the creation process for new weapons and facilitates rapid prototyping and balancing.

2.5 Bullet Time System

The bullet time mechanic represents one of "Spooky House's" most distinctive features, offering players superhuman reflexes inspired by "The Matrix" video game. This system creates dramatic slow-motion sequences while maintaining player agency and control.

2.5.1 Time Manipulation

The core of the bullet time system involves carefully managed time dilation (detailed in Appendix A.2):

- Adjustable time scale with configurable slowdown factor (currently 0.3x normal speed)
- Preservation of physics consistency through fixedDeltaTime adjustments
- Energy resource management to limit usage
- Automatic deactivation when energy is depleted
- Gradual energy recharge when not in use

This creates a tactical resource that players must manage, encouraging strategic activation during key moments rather than constant use.

2.5.2 Audiovisual Effects

To enhance the bullet time experience, the system implements comprehensive audiovisual changes:

- Screen tinting with configurable color to indicate the active state
- Global audio adjustments including volume reduction and pitch shifting
- Dedicated bullet time activation sound that plays at normal speed
- State restoration when bullet time ends

These effects create a dramatic shift in the game's presentation when bullet time is active, enhancing the sensation of heightened reflexes and perception.

2.5.3 Implementation Approach

The bullet time system uses several technical approaches to ensure smooth operation:

- Singleton pattern for global access from other systems
- Dictionary tracking of all audio sources for consistent sound management
- Unscaled time for energy depletion to ensure consistent drain rate
- Dynamically created UI elements for visual feedback

The system also includes safeguards to ensure time scale is always properly restored when the player exits bullet time, preventing potential game-breaking states where time remains permanently slowed.

A Technical Algorithms

This appendix contains the pseudocode implementations of key algorithms that power the core mechanics of "Spooky House."

A.1 Enemy AI State Machine

The following algorithm demonstrates the state-based decision making that drives enemy behavior in the game:

```
Result: Updated enemy state and behavior
while enemy is alive do
    currentState = GetCurrentState();
    switch currentState do
        case IDLE do
            UpdateAnimationState(false, false);
            agent.isStopped = true;
            waitTimer += Time.deltaTime;
            if waitTimer ¿ randomIdleTime then
                if canPatrol then
                    SetState(PATROL);
                end
            end
            if DistanceToPlayer() ¡ detectionRadius then
                SetState(CHASE);
            end
        end
        case PATROL do
            UpdateAnimationState(true, false);
            agent.isStopped = false;
            if !hasDestination then
                destination = GetRandomNavMeshPoint(position, patrolRadius);
                agent.SetDestination(destination);
            end
            if ReachedDestination() then
                SetState(IDLE);
            end
            if DistanceToPlayer() ¡ detectionRadius then
                SetState(CHASE);
            end
        end
        case CHASE do
            UpdateAnimationState(true, true);
            agent.isStopped = false;
            agent.SetDestination(playerPosition);
            if DistanceToPlayer() ¡= attackRange then
                SetState(ATTACK);
            end
            if DistanceToPlayer() ¿ losePlayerRadius then
                SetState(IDLE);
            end
        end
        case ATTACK do
            UpdateAnimationState(false, false);
            agent.isStopped = true;
            FaceTarget(playerPosition);
            if Time.time - lastAttackTime ¿ attackCooldown then
                PerformAttackSequence();
                lastAttackTime = Time.time;
            end
            if DistanceToPlayer() ¿ attackRange * 1.1 then
                SetState(CHASE);
            end
        end
    end
end
```

Algorithm 1: Enemy State Machine Algorithm

The algorithm implements a state machine controlling enemy behavior. Enemies transition between idle, patrol, chase, and attack states based on player proximity and timers. Each state has specific behaviors, like randomly patrolling when idle or attacking when the player is within range. The system integrates with Unity's NavMesh for pathfinding, enabling enemies to navigate complex environments while pursuing the player.

A.2 Bullet Time System

The signature bullet time feature is implemented through the following algorithm which carefully manages time scale, player physics, and environmental effects:

Result: Modified time scale with preserved player physics
Function `ToggleBulletTime(`*bool activate*`):`
  if *activate currentEnergy ¿ 0* **then**
    originalTimeScale = Time.timeScale;
    Time.timeScale = bulletTimeScale;
    Time.fixedDeltaTime = defaultFixedDeltaTime * Time.timeScale;
    isInBulletTime = true;
    PlayBulletTimeAudio();
    AdjustAllEnvironmentalSounds(loweredVolume, loweredPitch);
    ActivateVisualEffect(screenTint);
    // Apply player-specific physics preservation playerController.moveSpeed *= (1 / bulletTimeScale);
    playerController.jumpForce *= (1 / bulletTimeScale);
  end
  else
    Time.timeScale = 1.0f;
    Time.fixedDeltaTime = defaultFixedDeltaTime;
    isInBulletTime = false;
    StopBulletTimeAudio();
    RestoreAllEnvironmentalSounds();
    DeactivateVisualEffect(screenTint);
    // Restore normal player physics playerController.moveSpeed = defaultMoveSpeed;
    playerController.jumpForce = defaultJumpForce;
  end

Algorithm 2: Bullet Time Toggle Algorithm

This algorithm manages the game's signature bullet time mechanic. When activated (and if sufficient energy is available), it slows down the game's time scale while preserving the player's relative movement speed. The system adjusts audio (lowering volume and pitch of environmental sounds), applies visual effects (screen tinting), and modifies physics parameters to maintain responsive player control. When deactivated, all parameters are restored to normal values.

A.3 Wave Spawning System

Enemy waves are dynamically generated using the following algorithm that scales difficulty based on player progression:

Result: Progressive enemy spawning with difficulty scaling

Function `SpawnWave(`*int waveNumber*`):`

isSpawning = true;
// Calculate wave parameters with progressive difficulty baseEnemyCount = minEnemiesPerWave + (waveNumber * 0.5);
enemyCount = min(baseEnemyCount, maxEnemiesPerWave);
remainingEnemies = enemyCount;
// Calculate enemy type distribution standardEnemyChance = max(0.9 - (waveNumber * 0.05), 0.4);
specialEnemyChance = 1.0 - standardEnemyChance;
// Determine if boss wave isBossWave = (waveNumber % 5 == 0) (waveNumber ¿ 0);
if *isBossWave* **then**
    SpawnBossEnemy();
    remainingEnemies += 1;
end
// Spawn regular enemies **for** *i = 0* **to** *enemyCount - 1* **do**
    spawnPosition = GetValidSpawnPosition();
    enemyType = DetermineEnemyType(standardEnemyChance, specialEnemyChance);
    SpawnEnemy(enemyType, spawnPosition);
    yield WaitForSeconds(spawnInterval);
end
isSpawning = false;
nextWaveTime = Time.time + timeBetweenWaves;

Algorithm 3: Dynamic Wave Spawning Algorithm

This algorithm controls the game's wave-based enemy spawning system. It dynamically scales difficulty by increasing enemy numbers as waves progress and adjusting the likelihood of encountering special enemy types. Every fifth wave triggers a boss encounter for added challenge. The system ensures proper spacing between enemy spawns and validates spawn positions against the NavMesh to prevent enemies from appearing in invalid locations.

A.4 Player Movement System

The player's acrobatic movement capabilities, including the signature dive roll, are implemented as follows:

Result: Player movement with dive roll capabilities
Function `HandleJumpAndDive():`
jumpButtonPressed = Input.GetButtonDown("Jump");
if *jumpButtonPressed* **then**
if *isGrounded* **then**
// First jump velocity.y = jumpForce;
canDive = true;
lastJumpTime = Time.time;
PlayAnimation("jump");
end
else if *canDive (Time.time - lastJumpTime ¡ doubleJumpWindow)* **then**
// Dive roll on second press within window ExecuteDiveRoll();
end
end
Function `ExecuteDiveRoll():`
// Determine dive direction horizontalVelocity = new Vector3(controller.velocity.x, 0, controller.velocity.z);
if *horizontalVelocity.magnitude ¿ 0.5* **then**
diveDirection = horizontalVelocity.normalized;
end
else
diveDirection = GetInputDirection();
if *diveDirection.magnitude ¡ 0.1* **then**
diveDirection = transform.forward;
end
end
// Apply vertical boost velocity.y = jumpForce * 0.5;
// Start animation PlayAnimation("diveRoll");
isDiving = true;
canDive = false;
// During fixed update, apply: // controller.Move(diveDirection * (diveForce * Time.deltaTime))

Algorithm 4: Player Dive Roll System Algorithm

This algorithm manages the player's movement system, with particular focus on the signature dive roll mechanic. The system tracks the player's grounded state using raycasting for precision. When the jump button is pressed while grounded, the player executes a standard jump and starts a time window for dive roll eligibility. If the jump button is pressed again within this window, the player performs a dive roll in the direction of current movement or input. The dive roll provides both tactical advantages (evasion, positioning) and scoring benefits.

A.5 Combat Scoring System

The stylish combat scoring system evaluates player actions and assigns points based on the following algorithm:

Result: Combat style score with multipliers

Function `EvaluateKillStyle(`*Enemy enemy, DamageInfo damageInfo*`):`

```
baseScore = 100;
styleMultiplier = 1.0;
// Time since last kill affects combo timeSinceLastKill = Time.time - lastKillTime;
// Base multipliers if damageInfo.isHeadshot then
    styleMultiplier += 0.5;
end
if player.isInBulletTime then
    styleMultiplier += 0.3;
end
if player.isDiving then
    styleMultiplier += 0.6;
end
if player.isInAir then
    styleMultiplier += 0.4;
end
// Combo multiplier if timeSinceLastKill ¡ comboTimeWindow then
    killCombo += 1;
    comboMultiplier = min(1.0 + (killCombo * 0.1), 3.0);
end
else
    killCombo = 0;
    comboMultiplier = 1.0;
end
// Distance bonus distanceToEnemy = Vector3.Distance(player.position, enemy.position);
distanceMultiplier = 1.0 + (distanceToEnemy / 50.0);
// Max 1.5x at 25m
// Calculate final score finalScore = baseScore * styleMultiplier * comboMultiplier *
 distanceMultiplier;
// Update player state totalScore += finalScore;
lastKillTime = Time.time;
// Reward bullet time energy bulletTimeEnergy += finalScore * 0.05;
// Display style text DisplayStyleText(DetermineStyleRank(styleMultiplier));
return finalScore;
```

Algorithm 5: Style-Based Combat Scoring Algorithm

This algorithm implements the game's style-based scoring system, inspired by Devil May Cry's combo mechanics. It evaluates kills based on multiple factors: execution method (headshots), player state (in bullet time, diving, airborne), combo timing, and distance from target. Players are rewarded for chaining kills quickly with an escalating combo multiplier that can reach up to 3x. The system also incentivizes risk-taking by awarding higher scores for kills performed at greater distances. As a gameplay loop reinforcement, a portion of the score earned is converted to bullet time energy, encouraging players to perform stylish kills to maintain their bullet time capabilities.

A.6 Characters

Figure 1: Officer Edwards: A rookie cop with something to prove

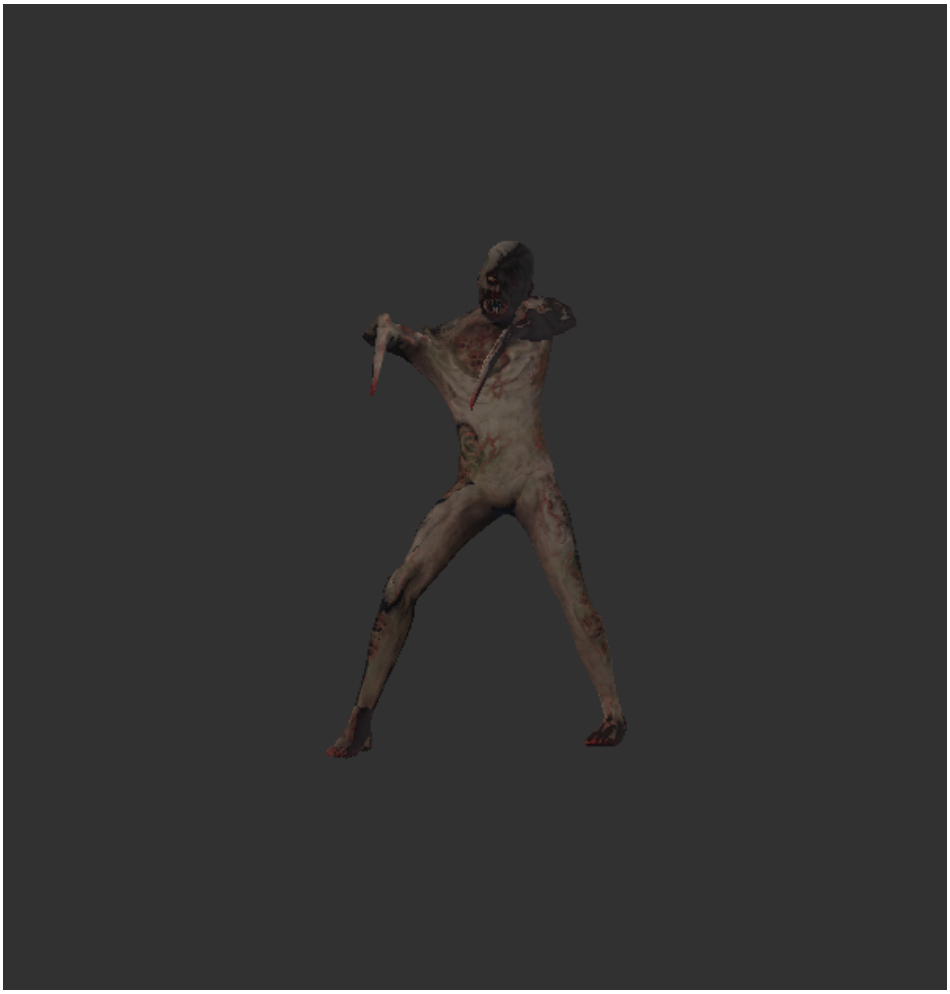

Figure 2: Reaver: Standard enemy

A.7 Player Animation States

The player character utilizes a state-based animation system with smooth transitions between key states. Figure 3 shows the primary animation states and their transitions.

The animation system uses boolean parameters and triggers in the Unity Animator to control transitions between states. The core animation states are:

- **Idle**: Default state when no input is detected
- **Run**: Active when movement input is detected
- **Shoot**: Triggered by left mouse button
- **Reload**: Activated by pressing R key
- **Jump/Dive**: Special movement states with jump triggered by space bar, and dive by pressing space again during the jump window

The animation controller works in concert with the player controller script to ensure visual feedback matches player input and game state. Transitions between states are managed through a combination of direct trigger calls and continuous parameter updates. The system is designed to handle animation blending for smooth transitions, particularly when changing from movement to combat actions and back.

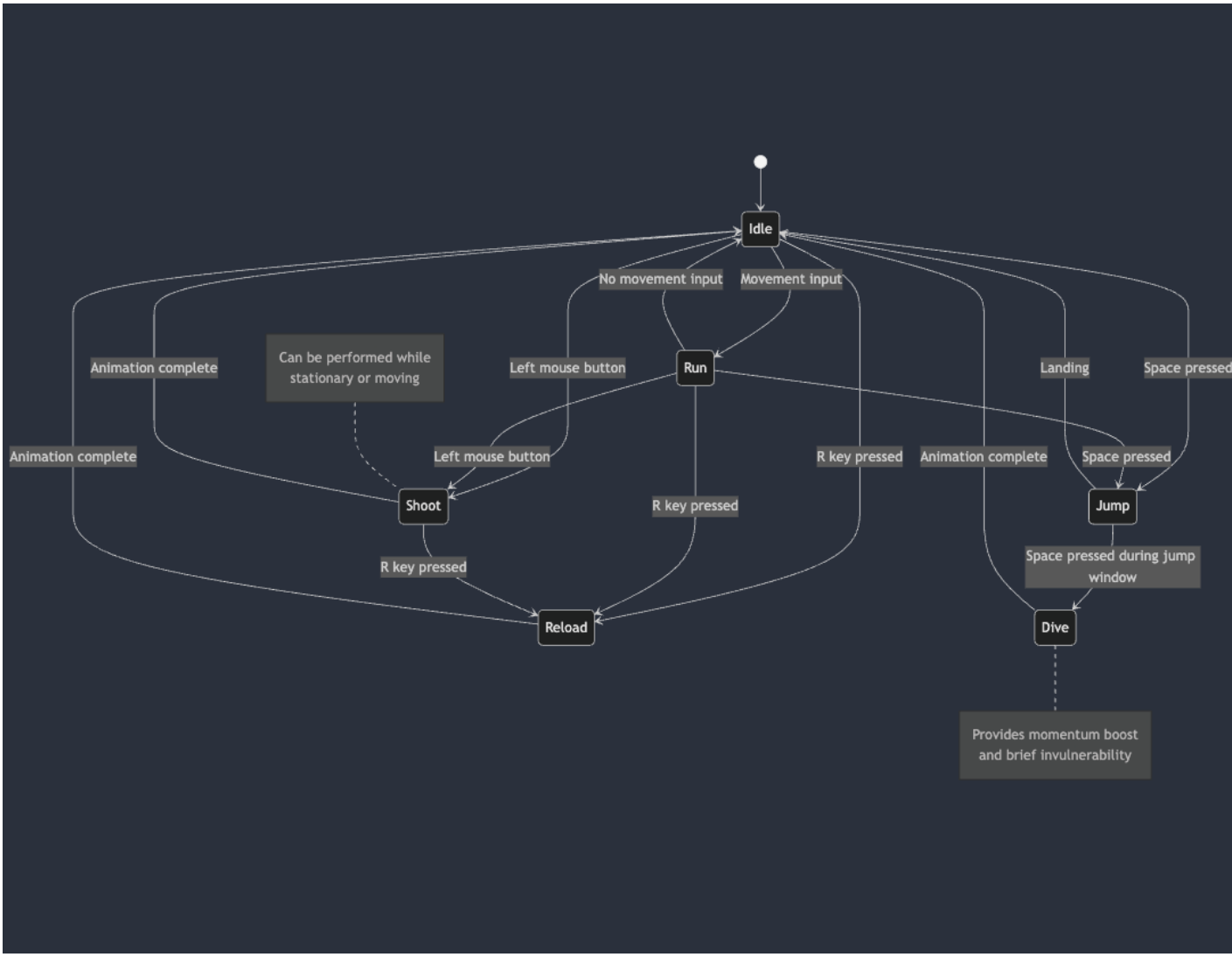

Figure 3: Player Animation State Machine

The dive roll animation is particuis larly important as it represents both a visual spectacle and a gameplay mechanic, providing brief invulnerability frames that skilled players can use to avoid damage. The reload animation similarly impacts gameplay by preventing other actions during its duration, creating tactical decisions around when to reload.