Technical Implementation Plan: Animation System Overhaul and State Machine Architecture for 'Binky Medieval Showdown'

1. Executive Summary and Architectural Vision

The development of "Binky Medieval Showdown" faces a critical inflection point where the initial prototype architecture—specifically the monolithic Player.ts controller—no longer supports the design requirements for advanced combat mechanics, character differentiation, and "juice" (visual feedback). The current implementation, characterized by a direct binding of input events to function calls, lacks the necessary abstraction to handle complex behaviors such as animation canceling, input buffering, and frame-perfect event synchronization.[1] This report outlines a comprehensive technical roadmap to refactor the game's core systems into a robust Finite State Machine (FSM) architecture using Phaser 3 and TypeScript.

The primary objective of this overhaul is to decouple the input logic from the animation and physics systems. By transitioning to an FSM, the development team can isolate the behaviors of the "Knight" and "Ninja" archetypes into discrete, manageable classes. This separation allows for the distinct "Game Feel" required by the design document: the Knight will operate as a high-mass, crowd-control-focused tank, while the Ninja will function as a high-velocity, evasion-oriented striker.[1] Furthermore, this plan integrates the "Ogre Pattern"—a decoupled hitbox management system currently used by enemy AI—into the player character, ensuring that all combat interactions are deterministic and visually synchronized.

This document provides an exhaustive analysis of the Phaser 3 API capabilities relevant to this transition, including the Event Emitter system, Animation State component, and WebGL rendering pipelines. It details the implementation of a hierarchical state machine, a command-based input buffer, and a revamped Skill Tree UI ("Book of Ascension") that leverages Phaser's camera and container systems for an immersive experience.

---

2. Core Architectural Analysis and Tech Stack Optimization

2.1 Limitations of the Current Monolithic Implementation

The existing Player.ts class functions as a "God Object," effectively managing input polling, physics updates, animation playback, and gameplay logic within a single scope. This violation

of the Single Responsibility Principle creates several technical debts. First, the input handling mechanism, described as binding pointerdown directly to a performAttack method, prohibits the implementation of fluid combat features like combos or secondary abilities because it cannot account for the player's current context.[1] For instance, a Knight mid-swing should not be able to instantly restart an attack animation upon a second click unless specific "cancel windows" are defined.

Second, the current system relies on arbitrary timers (e.g., setTimeout) to synchronize hitboxes with animations. This method is inherently brittle; if the frame rate fluctuates or the timeScale of an animation is altered for a slow-motion effect, the timer desynchronizes from the visual action, leading to "ghost hits" where damage registers before the weapon impacts the target.

2.2 Technology Stack and Framework Capabilities

The selected technology stack—Phaser 3.80+, TypeScript, and Vite [1]—provides a powerful foundation for the proposed overhaul. Phaser 3 differs significantly from its predecessor by utilizing a modular plugin system. The core Game instance manages global systems like the Renderer and Scene Manager, while individual Scenes maintain their own Update Lists and Display Lists.[1]

To support the overhaul, the architecture will leverage specific Phaser 3 components:

- **Update List & PreUpdate:** Custom Game Objects, such as our new StateMachine, must hook into the Scene's preUpdate cycle to process logic before rendering occurs.[1]
- **Animation Manager vs. Animation State:** We will utilize the distinction between global animation definitions and local animation states to share data between the Knight and Ninja while maintaining independent playback speeds and event listeners.[1]
- **Event Emitter:** The Phaser.Events.EventEmitter class will serve as the backbone for communication between the Input Manager, State Machine, and UI, ensuring loose coupling between systems.

2.3 Architectural Goals

The proposed architecture aims to achieve four key technical benchmarks:

1. **Determinism:** Combat outcomes must be strictly tied to animation frames via ANIMATION_UPDATE events rather than wall-clock time.[1]
2. **Scalability:** New skills defined in the SkillTreeData.ts must be addable without modifying the core character controller.[1]
3. **Responsiveness:** An input buffer must capture player intent within a 150ms window, preventing "dropped" inputs during high-intensity combat.
4. **Visual Fidelity:** The system must support "Juice" through the integration of Phaser's WebGL FX pipelines (Glow, Shadow, ColorMatrix) triggering dynamically based on state transitions.[1]

---

3. The Finite State Machine (FSM) Architecture

The Finite State Machine represents the central nervous system of the new PlayerController. Unlike the previous boolean-heavy logic (e.g., if (isAttacking &&!isStunned)), the FSM allows the entity to exist in exactly one state at a time, with clearly defined transitions. This rigidity is advantageous for action games where states like Idle, Attack, and Stunned are mutually exclusive.

3.1 Interface Design and Polymorphism

To ensure type safety and modularity in TypeScript, all states will implement a strict interface. This contract guarantees that the State Machine can manage any state—whether it be a standard MoveState or a complex, character-specific WhirlwindState—without knowing the implementation details.

TypeScript

```
export interface IState {
    enter(data?: any): void;
    execute(time: number, delta: number): void;
    exit(): void;
    handleInput(input: InputCommand): void;
}
```

The lifecycle methods serve distinct purposes in the Phaser ecosystem:

- **enter():** This method initializes the state. It is responsible for triggering the Phaser animation (e.g., sprite.play('knight_idle')) and configuring initial physics properties. For example, entering a BlockState would immediately set body.setVelocity(0) to root the character.
- **execute(time, delta):** This method runs every frame within the Scene's update loop. It handles continuous logic, such as applying drag to velocity, checking for ground collision via body.blocked.down, or interpolating rotation. The delta parameter is crucial for frame-rate independent calculations, ensuring movement remains consistent regardless of the user's hardware.[1]
- **exit():** This cleanup method is vital for preventing memory leaks and logic errors. It must remove event listeners (e.g., sprite.off('animationcomplete')) and destroy temporary objects like hitboxes or particle emitters created during the state.
- **handleInput():** This method delegates decision-making to the specific state. While the IdleState might transition to JumpState upon receiving a JUMP command, the StunnedState would simply ignore it.

3.2 Hierarchical State Inheritance

To reduce code duplication, the system will employ a hierarchical class structure. A BaseState abstract class will provide common utility references (to the Scene, Player, and SoundManager). Subclasses will group shared behaviors.

- **GroundState:** Handles logic common to all ground-based actions, such as applying friction and transitioning to FallState if the player walks off a ledge. IdleState and MoveState inherit from this.
- **AirState:** Manages air resistance and landing detection. JumpState and FallState inherit from this.
- **CombatState:** Manages combat flags and prevents standard movement. AttackState and SkillState inherit from this.

This hierarchy allows for global rules. For instance, GroundState can enforce a rule that "taking damage transitions to HitReactionState," which automatically propagates to Idle, Move, and Crouch without redundant code.

3.3 The State Machine Controller

The StateMachine class acts as the manager. It maintains a registry of available states and handles the actual switching logic.

Property	Type	Description
states	Map<string, IState>	A dictionary mapping string keys (e.g., 'idle') to state instances.
currentState	IState	The reference to the active state logic.
context	PlayerController	The generic entity being controlled.
isChanging	boolean	A lock flag to prevent recursive state transitions during a single frame.

The transition logic within changeState(key) is critical. It must first verify the key exists, call currentState.exit(), update the reference, and finally call newState.enter(). This sequence ensures that the previous state's cleanup (e.g., disabling a hitbox) always happens before the new state's initialization (e.g., starting a move).

---

4. Input System Overhaul: Buffering and Decoupling

The analysis of Player.ts revealed a rigid coupling between input detection and action. To support the diverse requirements of the Knight and Ninja, the input system must be abstracted into a command-based architecture.

4.1 The Command Pattern

Instead of checking for specific keys (e.g., keys.W.isDown) inside the player logic, we introduce an InputManager that translates hardware events into semantic game commands. This allows for easy remapping and multi-device support (Keyboard vs. Gamepad).

Input Mapping Table:

Hardware Input	Semantic Command	Context (Knight)	Context (Ninja)
KeyW / Up	MOVE_UP	Walk Up	Run Up
Space	MOBILITY_ACTION	Block / Shield Raise	Dash / Teleport
MouseLeft	PRIMARY_ATTACK	Sword Swing	Dagger Slash
MouseRight	SECONDARY_ATTACK	Heavy Slam	Shuriken Throw
KeyQ	SKILL_1	Skill Tree Ability 1	Skill Tree Ability 1

4.2 The Input Buffer

Action games require high responsiveness. If a player presses "Attack" 50ms before their current attack finishes, the game should queue that input and execute it immediately upon state completion. The InputBuffer class will store these commands with a timestamp.

TypeScript

```
class InputBuffer {
   private buffer: { command: Command, timestamp: number } =;
   private readonly BUFFER_WINDOW = 150; // ms

   public push(command: Command): void {
      this.buffer.push({ command, timestamp: Date.now() });
   }

   public getLatest(): Command | null {
      const now = Date.now();
      // Filter expired commands
      this.buffer = this.buffer.filter(item => now - item.timestamp < this.BUFFER_WINDOW);
      return this.buffer.length > 0? this.buffer.shift().command : null;
   }
}
```

The State Machine's execute method will query inputBuffer.getLatest() at the start of every frame. If valid input exists, the currentState.handleInput() method determines if a transition is valid. For example, a RecoveryState (the end of an attack) might accept a PRIMARY_ATTACK command to transition into the second hit of a combo, whereas a StunnedState would consume and ignore the command.

---

5. Animation Event Synchronization

The most significant technical challenge identified is the desynchronization of gameplay events from animations. The current setTimeout approach fails because it operates on wall-clock time, while animations operate on game loop frames. The solution utilizes Phaser 3's robust Animation Event system.

5.1 Frame-Based Logic

Phaser animations emit events that are perfectly synced to the rendering cycle. We will leverage Phaser.Animations.Events.ANIMATION_UPDATE to drive combat logic.[1]
When the AttackState is entered:

1. The animation plays: sprite.play('knight_slash').
2. A listener is attached: sprite.on('animationupdate', this.onFrameUpdate, this).

Inside onFrameUpdate(anim, frame, gameObject), we inspect the frame.index. The configuration for each attack (loaded from JSON) will specify a damageFrame.

TypeScript

```
if (frame.index === this.attackConfig.damageFrame) {
   this.triggerHitbox();
   this.playSwipeSound();
   this.spawnVFX();
}
```

This guarantees that the hitbox appears exactly when the sword is fully extended, regardless of frame rate drops or slow-motion effects.

5.2 The "Ogre Pattern" Integration

The research indicates that the Ogre enemy uses a MeleeAttack.ts object to manage hitboxes.[1] This is a "Fire-and-Forget" pattern that we will standardize for the player. When triggerHitbox() is called, the Player does not calculate damage. Instead, it instantiates or wakes a SkillObject from a pool.
The Skill Object Lifecycle:

1. **Spawn:** Placed at an offset relative to the player (e.g., 50px in front).
2. **Activate:** Enables its Arcade Physics body (Circle or Rectangle).[1]
3. **Overlap:** Checks for overlaps with the EnemyGroup using scene.physics.overlap().
4. **Execute:** Applies damage and knockback to overlapping enemies.
5. **Disable:** Auto-destroys or returns to pool after a short duration (e.g., 100ms).

This separation of concerns allows the Player Controller to focus on movement and state, while the SkillObject handles the specifics of collision and damage application.

6. Archetype Differentiation: Knight vs. Ninja

To satisfy the requirement of differentiating the Knight and Ninja, we will utilize the State Machine to enforce distinct physics and behavior rules. The "feel" of a character is largely derived from their acceleration, friction, and state transition timings.

6.1 Physics Configuration (Arcade Physics)

Phaser's Arcade Physics engine allows for fine-tuning of movement dynamics through properties like drag, acceleration, and maxVelocity.[1]

Table 1: Archetype Physics Parameters

Parameter	Knight (Tank/Bruiser)	Ninja (Speedster)	Implication
mass	High	Low	Determines knockback resistance.
drag.x	2000 (High)	500 (Low)	Knight stops instantly; Ninja slides slightly.
acceleration	300 (Slow ramp)	1000 (Instant)	Knight feels heavy; Ninja feels responsive.
maxVelocity	150	400	Ninja moves significantly faster.
bounce	0.1	0.0	Knight might have slight recoil; Ninja has none.

6.2 Unique State Mechanics

The Knight: The Juggernaut
The Knight's combat design focuses on "Defensive Offense." We will implement a ShieldBlockState.

- **Input:** Holding MOBILITY_ACTION (Space).
- **Logic:** The state sets player.body.setVelocity(0). It enables a generic onDamage hook that reduces incoming damage values by 80%.
- **VFX:** Uses Phaser.Actions.PlaceOnCircle to position shield particle effects around the player.[1]
- **Attacks:** Attacks have "Super Armor," meaning the HitReactionState cannot interrupt the AttackState during the active frames.

The Ninja: The Glass Cannon
The Ninja focuses on evasion and burst. We will implement a DashState.

- **Input:** Pressing MOBILITY_ACTION (Space).
- **Logic:** Sets velocity to 800 (double max run speed). Temporarily disables enemy collision using body.checkCollision.none = true to allow passing through mobs.

- **VFX:** Spawns "After-image" sprites using a Blitter object for high performance, creating a trail effect.[1]
- **Animation Canceling:** The Ninja's AttackState will monitor the Input Buffer during the "Recovery" phase. If a movement command is detected, it immediately transitions to MoveState, cancelling the back-swing animation. The Knight does not have this privilege.

---

7. The Skill System and Skill Tree UI

The "Book of Ascension" represents the progression system. This requires a robust UI implementation that interacts with the underlying game data.

7.1 Data-Driven Skill Definitions

The SkillTreeData.ts will be expanded to support dynamic modifiers. Instead of hardcoded values, nodes will return modifier objects.

TypeScript

```
interface SkillNode {
  id: string;
  name: string;
  type: 'stat' | 'ability';
  modifiers: {
    stat?: string; // e.g., 'walkSpeed'
    value?: number; // e.g., +50
    unlockState?: string; // e.g., 'WhirlwindState'
  };
  position: { x: number, y: number }; // For UI layout
}
```

7.2 UI Implementation with Phaser Cameras

To achieve the "zooming and panning" requirement [1], the Skill Tree should exist in a separate Scene (SkillTreeScene) that runs in parallel or on top of the Game Scene. This scene will utilize a specialized Camera.

- **Panning:** We will enable drag input on the background: this.input.on('drag', (pointer, gameObject, dragX, dragY) => { camera.scrollX -= dragX; camera.scrollY -= dragY; });.[1]
- **Zooming:** We will map the mouse wheel to the camera zoom: this.input.on('wheel', (pointer, over, deltaX, deltaY, deltaZ) => { camera.setZoom(Math.clamp(camera.zoom - deltaY * 0.001, 0.5, 2)); });.[1]

- **Connections:** We will use Phaser.GameObjects.Graphics to draw lines between nodes. These lines must be redrawn in the update loop if the nodes move, or drawn once to a Texture if static.

7.3 Visualizing Nodes

Each node in the tree will be a Phaser.GameObjects.Container holding:

1. **Sprite:** The icon background.
2. **Image:** The skill icon.
3. **Text:** The skill level (e.g., "0/3").
4. **VFX:** A PostFX Glow effect (WebGL) that activates when the mouse hovers over the node, utilizing gameObject.postFX.addGlow().[1]

---

8. Visual Effects ("Juice") and Rendering Pipelines

To meet the "modern, immersive" requirement, visual feedback must be integral to the State Machine transitions. We will utilize Phaser's WebGL Pipeline capabilities to create distinctive effects for each archetype.

8.1 Post-Processing FX

Phaser 3's FX system allows for shaders to be applied to Game Objects.

- **Damage Flash:** When entering HitReactionState, we will apply a ColorMatrix effect to the sprite, setting brightness to max to create a white flash sprite.preFX.addColorMatrix().brighten(255).[1]
- **Ninja Speed:** During DashState, we apply a Blur effect or Barrel distortion to the camera to sell the sensation of speed.[1]
- **Knight Rage:** If the Knight unlocks a "Berzerker" skill, we can apply a red Glow effect to the character sprite using sprite.postFX.addGlow(0xff0000).[1]

8.2 Particle Systems

We will use Phaser.GameObjects.Particles.ParticleEmitter for state-specific effects.

- **Movement:** The MoveState will trigger a "Dust" emitter at the character's feet (y + height/2) whenever velocity is non-zero.
- **Impact:** The SkillObject (hitbox) will spawn a one-shot "Blood" or "Spark" emitter upon overlap with an enemy.
- **Death:** Upon death, we can use the Explode emitter configuration to shatter the sprite into constituent parts.

---

9. Migration Strategy and Implementation Roadmap

Refactoring a live codebase requires a surgical approach to avoid destabilizing existing

features.

Phase 1: Core Systems Implementation

1. **Create StateMachine.ts and State.ts:** Implement the base classes and interfaces.
2. **Create InputManager.ts:** Implement the InputBuffer and command mapping logic.
3. **Refactor Player.ts:** Strip out the update() logic and replace it with stateMachine.execute(). Initialize the StateMachine with just IdleState and MoveState. Verify movement works with the new physics parameters.

Phase 2: Combat Integration

1. **Create AttackState.ts:** Move the performAttack logic into this state.
2. **Implement Animation Sync:** Replace setTimeout with animationupdate event listeners.
3. **Refactor SkillSystem:** Implement the factory pattern for SkillObject creation (Ogre Pattern). Ensure AttackState calls the factory on the damage frame.

Phase 3: UI and Progression

1. **Build SkillTreeScene:** Implement the camera panning/zooming and node rendering.
2. **Connect Data:** Load SkillTreeData.ts and ensure unlocking a node updates the PlayerController's allowed states (e.g., unlocking Dash allows Input.DASH to trigger DashState).

Phase 4: Polish and Juice

1. **Integrate VFX:** Add enter/exit hooks in states to trigger Particle Emitters and WebGL FX.
2. **Tune Archetypes:** Finalize the physics values for Knight and Ninja to ensure they feel distinct.

---

10. Conclusion

This technical implementation plan provides a clear path to transforming "Binky Medieval Showdown" into a mechanically deep and visually responsive game. By adopting a State Machine architecture, we solve the critical issues of input responsiveness and animation synchronization. The separation of Knight and Ninja logic into distinct states and physics configurations ensures character diversity, while the integration of the "Ogre Pattern" ensures a robust and scalable combat system. Leveraging Phaser 3's advanced features—specifically Animation Events, Arcade Physics, and WebGL FX—will deliver the high-quality "game feel" targeted in the research objectives.

---

11. Detailed Technical Specifications & API Usage

11.1 Managing Game Object Groups and Pooling

To maintain performance with the new "Skill Object" pattern, we must avoid creating and destroying objects repeatedly, which triggers Garbage Collection spikes. We will utilize Phaser.GameObjects.Group for object pooling.[1]

TypeScript

```
// In PlayerSkillSystem
this.hitboxPool = this.scene.add.group({
   classType: Phaser.GameObjects.Zone,
   maxSize: 20,
   runChildUpdate: true
});

public getHitbox(): Phaser.GameObjects.Zone {
   let hitbox = this.hitboxPool.get();
   if (!hitbox) return; // Pool empty
   hitbox.setActive(true);
   hitbox.setVisible(true);
   return hitbox;
}
```

This pattern is essential for the Ninja's multi-hit attacks or projectile shurikens, where object turnover is high.

11.2 Tween Management for UI

For the "Book of Ascension," transitions (opening tooltips, unlocking nodes) should be handled via Phaser.Tweens.TweenManager.

- **Tooltip Open:** Use an Elastic ease to make the tooltip pop out.
  JavaScript
  ```
  scene.tweens.add({
     targets: tooltipContainer,
     scaleX: 1,
     scaleY: 1,
     duration: 300,
     ease: 'Back.out'
  });
  ```

  This adds the requested "Juice" to the UI interactions.[1]

11.3 Blitter for Environmental Decals

The Knight's heavy attacks are described as having "weight." To visualize this, ground slams should leave cracks. Using standard Sprites for permanent decals is expensive. Instead, we will use a Phaser.GameObjects.Blitter.[1]

- **Usage:** A single Blitter object handles all ground crack textures.
- **Implementation:** On the impact frame of a "Ground Slam" attack, the AttackState calls decalBlitter.create(x, y, 'crack_texture'). Because Blitter objects are not individually updated or physics-enabled, hundreds can be rendered with minimal performance cost, permanently altering the battlefield visuals.

11.4 Timeline for Cutscene/Cinematic Attacks

For "Ultimate" abilities mentioned in the target state (e.g., a screen-clearing Knight attack), simple state logic might become messy. We can utilize Phaser.Time.Timeline.[1]

- **Structure:** A Timeline allows scheduling commands:
  JavaScript
  ```
  const timeline = scene.add.timeline();
  timeline.play();
  ```

This allows for complex, cinematic sequences that are tightly timed without polluting the update loop with timer variables.

11.5 Arcade Physics: Precision Movement

To differentiate the movement feel:

- **Knight:** We rely on body.setDrag(x, y) to slow him down. High drag means he resists changing direction, feeling heavy.
- **Ninja:** We rely on body.setAcceleration(x, y) rather than setting velocity directly. This allows for "curving" movement arcs. However, for the "Dash," we specifically set velocity directly to override momentum.
- **Collision:** We must use body.setSize() and body.setOffset() dynamically. When the Ninja enters CrouchState or RollState, the hitbox height must be reduced to allow passing under projectiles. This is done in the enter() method of the state and restored in exit().[1]

By rigorously applying these patterns, the "Binky Medieval Showdown" codebase will evolve from a prototype to a professional-grade system capable of supporting extensive future content.

Works cited

1. GEMINI_DEEP_RESEARCH_PLAN.txt