A Technical Deep Dive into the Design and Implementation of a High-Powered Boomerang Weapon

Section 1: The Philosophy of a Game Boomerang: Control, Feel, and Physics

The creation of a compelling in-game weapon, particularly one as dynamic as a high-powered boomerang, is an exercise in design, not a pursuit of perfect simulation. The process begins not with code, but with a foundational philosophy that prioritizes player experience, control, and "game feel" above all else. Before a single line of code is written, a developer must make critical choices about the nature of the weapon's physics and the degree of agency the player will have over its behavior. These decisions will define the weapon's role within the game's ecosystem, shaping everything from combat encounters to level design.

1.1 Deconstructing "Real" Physics: Why Simulation is the Enemy of Fun

A real-world boomerang's flight is a product of extraordinarily complex aerodynamic and gyroscopic principles. Its curved path is generated by the interplay of lift, drag, and gyroscopic precession. As the boomerang spins, the wing moving forward into the air travels faster than the wing moving backward, generating differential lift.[1] This imbalance creates a torque that, instead of flipping the boomerang over, causes its axis of rotation to precess, gradually turning its path into a circle.[1] The flight is further complicated by factors like the angle of attack, initial rotational and translational velocities, and even air density.[3]

A direct simulation of this intricate system within a 2D platformer is not only computationally expensive but fundamentally undesirable from a gameplay perspective. The chaotic nature of real-world boomerang flight means that minuscule variations in the initial throw conditions—parameters a player could never perfectly replicate—result in dramatically different trajectories.[4] This inherent unpredictability would render the weapon unreliable for the precise actions required in a platformer, such as hitting a specific enemy or a distant switch. It would frustrate the player, who would feel a lack of mastery and control over a core mechanic.

The solution lies in the deliberate abstraction of these physical principles into a simplified,

controllable "game physics" model. The objective is not realism, but *believability* and, more importantly, *utility*. The design should capture the *essence* of a boomerang—its arcing path and return capability—while discarding the complexities that hinder gameplay. This approach aligns with the core tenets of platformer design, where even a fundamental action like jumping is heavily stylized to grant the player a level of mid-air control that is physically impossible but essential for satisfying gameplay.[6] The physics-driven platformer, for instance, does not rely on true Newtonian simulation but on a curated set of rules that allow for emergent, yet learnable, outcomes.[6] The high-powered boomerang must follow this same design philosophy.

1.2 The Core Design Choice: Path-Defined vs. Force-Based Motion

The first and most significant technical decision is how the boomerang's movement will be calculated. This choice falls into two broad categories, each with profound implications for the weapon's feel and function.

Kinematic (Path-Defined) Motion

In a kinematic approach, the boomerang's trajectory is explicitly defined by a mathematical curve. The designer, and in some cases the player, has direct and absolute control over the path. The weapon's position at any given time is simply a point calculated along this pre-determined curve. This method offers perfect predictability and reliability. A developer can use parametric equations for simple arcs or, for more artistic control, employ Bézier splines to define complex, stylized paths.[7] Because the path is known in advance, this approach is exceptionally well-suited for games with puzzle elements, where the player might need to thread the boomerang through a specific sequence of openings or hit a series of switches in a precise order. The weapon becomes less of a projectile and more of a remote-controlled tool, where the player's skill is expressed in the initial planning and aiming of the throw.

Dynamic (Force-Based) Motion

In a dynamic approach, the boomerang is treated as a physics object governed by a set of simplified, simulated forces. At the moment of the throw, an initial impulse force is applied to give it velocity. Subsequently, continuous forces—such as a constant downward pull for gravity and, after a certain condition is met, a "return" force that accelerates it back toward the player—are applied in each frame of the game loop. The final trajectory is not pre-calculated but emerges from the interaction of these forces with each other and with the game world's physics. This method results in more organic, less predictable behavior. It can

create a higher skill ceiling, as players learn to manipulate the forces to their advantage, for example, by using a collision with a wall to alter the return trajectory. This philosophy mirrors that of "physics-driven platforming," where the joy comes from mastering a dynamic system rather than executing a pre-solved puzzle.[6] The choice between these two models is a statement of design intent: a path-defined boomerang serves a game of precision and puzzles, while a force-based boomerang serves a game of emergence and physical mastery.

1.3 Player Agency: "Committed" vs. "Variable" Throws

Drawing a parallel to the fundamental concepts of jumping in platformer design, the boomerang's control scheme can be categorized as either "committed" or "variable".[6] This choice determines the level of player influence after the weapon has been thrown and is a critical factor in its overall feel.

Committed Throw

A committed throw is one where, once the boomerang leaves the player's hand, its path and behavior are fixed until it returns. The player has no further input. This approach, much like committed jumping, is simple for players to understand; the outcome of a throw is immediately clear.[6] It places all the emphasis on the initial aim and timing. However, it can also feel rigid and unforgiving if the player makes a mistake or if the tactical situation changes mid-throw.

Variable Throw

A variable throw grants the player continued agency over the boomerang while it is in flight. This can be implemented in several ways, each adding a layer of strategic depth:

- **Early Recall:** The player can press a button to interrupt the outbound path and immediately trigger the return state. This allows for tactical flexibility, letting the player "shorten the leash" on the weapon to react to new threats or correct a poorly aimed throw.
- **Path Manipulation:** The player might be able to apply a subtle "nudge" force, slightly altering the boomerang's trajectory mid-flight. This adds a high-skill element of fine control.
- **Apex Hang:** Holding the throw button could cause the boomerang to hover or spin in place at the apex of its path. Releasing the button would then initiate the return. This transforms the weapon into a delayed-action tool, perfect for timing-based puzzles or setting up traps for enemies.

A hybrid approach often yields the most satisfying results. For instance, a boomerang could follow a predictable, "committed" path on its way out, but the player retains the "variable"

ability to recall it at any time. This provides the reliability needed for precision tasks while still giving the player a crucial tool for adapting to dynamic situations, striking a superior balance between usability and expressive depth.

Section 2: Architectural Blueprint: A State Machine for Boomerang Logic

To manage the boomerang's multifaceted behavior—moving away, pausing, returning, being held—a robust software architecture is not a luxury but a necessity. Attempting to control these distinct phases of operation within a single update loop using a complex web of conditional statements (if/else) will inevitably lead to fragile, unmanageable "spaghetti code." The industry-standard solution for this class of problem is the Finite State Machine (FSM), a design pattern that provides a clean, scalable, and intuitive framework for modeling an object's behavior.[10]

2.1 Why a Finite State Machine (FSM) is Essential

An FSM is a model of computation based on a hypothetical machine that can be in exactly one of a finite number of *states* at any given time. The machine can change from one state to another in response to external inputs or internal conditions; this change is called a *transition*. For the boomerang, this pattern is a perfect fit. Each phase of its life cycle can be encapsulated in a distinct state. The logic for movement, collision detection, and animation is contained entirely within the current state. This modularity is the FSM's greatest strength. To change how the boomerang behaves on its return trip, one only needs to modify the INBOUND state's code, without any risk of unintentionally breaking the logic of the OUTBOUND state. This makes the system far easier to debug, maintain, and, most importantly, expand upon.

2.2 Defining the States and Their Responsibilities

A well-designed FSM for a high-powered boomerang would consist of the following core states:

HELD

This is the default, inactive state.

- **Responsibilities:** The boomerang's position is locked to a spawn point on the player

character (e.g., the hand). It is not visible or is rendered as part of the player's sprite. The state's primary logic is to listen for the player's "throw" input.
- **Transitions:** Transitions to the OUTBOUND state upon receiving the throw command.

OUTBOUND

This state governs the boomerang's initial throw and outward journey.
- **Responsibilities:** Upon entering this state, the boomerang is instantiated in the game world at the player's spawn point. Its movement is governed by one of the mathematical models discussed in Section 3 (e.g., a parametric arc or a Bézier curve). It actively checks for collisions with objects on the Enemies and WorldGeometry collision layers. It also tracks the conditions that will trigger its return, such as elapsed time or distance traveled.
- **Transitions:** Transitions to the APEX or INBOUND state when a return condition is met (e.g., distance_traveled > max_range).

APEX (Optional State)

This is a brief, transitional state that can be inserted between the outbound and inbound phases to add stylistic flair and a feeling of power.
- **Responsibilities:** The boomerang's forward motion ceases. It might hover in place, spin with a different visual effect, or perform a wide, dramatic turn. This state is an ideal place to trigger unique sound effects and particle systems that emphasize the moment of transition, providing clear feedback to the player that the weapon is about to return.
- **Transitions:** Transitions to the INBOUND state after a short, fixed duration (e.g., 0.25 seconds).

INBOUND

This state manages the boomerang's return to the player.
- **Responsibilities:** The movement logic switches to a homing algorithm (detailed in Section 4) that targets the player's current position. The collision response logic may also change; for example, it might now pass harmlessly through enemies it has already damaged, or it might be able to hit them a second time. It actively checks for a collision with the player's "catch" hitbox.
- **Transitions:** Transitions to the HELD state upon collision with the player.

2.3 State Transitions: The Rules of Behavior Change

The transitions are the connective tissue of the FSM, defining the precise rules that govern when and how the boomerang changes its behavior.

- **HELD → OUTBOUND:**
  - **Trigger:** Player presses the Attack button.
  - **Action:** Instantiate the boomerang object, set its initial velocity (factoring in player momentum), and switch its state to OUTBOUND.
- **OUTBOUND → INBOUND (or APEX):**
  - **Triggers (any of the following):**
    - **Distance-Based:** if (distance_from_player > max_range)
    - **Time-Based:** if (time_in_state > max_duration)
    - **Collision-Based:** if (collides_with_impassable_wall)
    - **Player-Initiated:** if (player_presses_recall_button)
  - **Action:** Change the state to INBOUND (or APEX) and initiate the new state's movement logic.
- **APEX → INBOUND:**
  - **Trigger:** if (time_in_state > apex_hang_time)
  - **Action:** Change the state to INBOUND.
- **INBOUND → HELD:**
  - **Trigger:** Collision with the player's designated "catch" hitbox.
  - **Action:** Deactivate or destroy the boomerang object, reset the player's throw cooldown, and set the internal state variable back to HELD.

This FSM structure is not merely a method for organizing code; it is a powerful design tool. By simply adding new states or modifying the transition rules, a developer can create a vast array of different weapon types from a single, unified architecture. For example, a "sticky" boomerang that embeds itself in a wall could be created by adding a STUCK state, with a transition from OUTBOUND on wall collision and a transition back to INBOUND after a timer or player input. The FSM directly translates design ideas into a modular and extensible code implementation.

Section 3: The Outward Journey: Mathematical Modeling of the Throw

The OUTBOUND state is where the boomerang's character is first defined. The choice of mathematical model for its trajectory dictates its function, its feel, and the types of gameplay scenarios it can support. A developer should select a model not based on a desire for realism, but on the intended gameplay loop: direct combat, intricate puzzle-solving, or area denial.

3.1 The Parametric Arc: Classic Projectile Motion

This model treats the boomerang as a standard projectile under the influence of gravity,

resulting in a familiar parabolic arc. It is the most common and intuitive approach for action-oriented projectiles.

The Math

The position of the projectile at any given time, t, is described by a pair of parametric equations that separate the horizontal (x) and vertical (y) motion [13]:
$x(t)=v0x \cdot t$
$y(t)=v0y \cdot t - 21g \cdot t2$

Where:

- (x(t),y(t)) is the position at time t.
- v0x and v0y are the initial horizontal and vertical components of the velocity.
- g is the acceleration due to gravity, a tunable constant that determines the "weight" of the projectile.

Implementation

The key to a successful implementation is calculating the initial velocity vector, (v0x,v0y). This is determined by the throw's base power and angle, but critically, it should also be influenced by the player's own momentum to feel responsive and connected.[15]

1. **Base Velocity:** First, calculate the base throw velocity from a designer-defined angle (θ) and speed (S).
    - base_velocity_x = S \cdot \cos(\theta)
    - base_velocity_y = S \cdot \sin(\theta)
2. **Player Momentum:** Add a fraction of the player's current velocity. The momentum_factor is a crucial "feel" variable; a value of 1.0 means the boomerang inherits 100% of the player's momentum, while 0.0 means it ignores it completely.
    - final_v0x = base_velocity_x + (player.velocity.x \cdot momentum_factor)
    - final_v0y = base_velocity_y + (player.velocity.y \cdot momentum_factor)
3. **Update Loop:** In the game's update loop, the position is updated each frame. A time_in_flight variable is incremented by the frame's delta time.

```
// Pseudocode for Parametric Arc Update
class Boomerang {
   Vector2 initial_position;
   Vector2 initial_velocity;
   float gravity;
   float time_in_flight = 0;

   void Throw(Vector2 start_pos, Vector2 start_vel) {
      this.initial_position = start_pos;
```

```
        this.initial_velocity = start_vel;
        this.time_in_flight = 0;
    }

    void Update(float delta_time) {
        time_in_flight += delta_time;
        position.x = initial_position.x + initial_velocity.x * time_in_flight;
        position.y = initial_position.y + initial_velocity.y * time_in_flight - 0.5 * gravity *
time_in_flight^2;
    }
}
```

3.2 The Stylized Curve: Bézier Splines for Artistic Control

For a boomerang that needs to follow a highly controlled, artistic, or non-physical path, Bézier splines are the ideal tool. They allow the designer or player to define a smooth curve using a set of control points.[8] A quadratic Bézier curve is sufficient for most boomerang paths.

The Math

A quadratic Bézier curve is defined by a start point (P0), an end point (P2), and a single control point (P1) that pulls the curve towards it. The position on the curve, B(t), for a parameter t ranging from 0 to 1, is given by:

$$B(t)=(1-t)2P0+2(1-t)tP1+t2P2$$

Implementation

The implementation challenge lies in intelligently placing the three control points to generate the desired path.[7]

1. **Define Control Points:**
    - P0 (Start Point): The player's position at the moment of the throw.
    - P2 (End Point): The desired maximum range of the boomerang. This can be calculated as a point along a vector from the player, based on the throw angle and a max_range variable.
    - P1 (Control Point): This point defines the arc's shape. It can be calculated by taking the midpoint between P0 and P2, and then displacing it perpendicularly. The direction and magnitude of this displacement determine the curve's height and direction, allowing for high, looping throws or flat, direct ones. This

displacement can even be influenced by player input (e.g., holding 'up' creates a higher arc).

2. **Update Loop:** The boomerang's movement involves interpolating along the curve over a set duration.

```
// Pseudocode for Bézier Curve Update
class Boomerang {
   Vector2 p0, p1, p2;
   float total_duration;
   float time_on_curve = 0;

   void Throw(Vector2 start_p, Vector2 control_p, Vector2 end_p, float duration) {
      this.p0 = start_p;
      this.p1 = control_p;
      this.p2 = end_p;
      this.total_duration = duration;
      this.time_on_curve = 0;
   }

   void Update(float delta_time) {
      time_on_curve += delta_time;
      float t = time_on_curve / total_duration;
      if (t > 1.0) { t = 1.0; }

      // Bézier formula
      float one_minus_t = 1.0 - t;
      position = (one_minus_t^2 * p0) + (2 * one_minus_t * t * p1) + (t^2 * p2);
   }
}
```

3.3 Circular and Elliptical Paths for Unique Trajectories

For a truly unique, high-powered feel, the boomerang can follow a circular or elliptical path. This is less about precision targeting and more about area denial or hitting multiple targets in a wide sweep.

The Math

These paths are also defined parametrically, using trigonometric functions [19]:

- Circle:

$x(t) = C_x + r \cdot \cos(\omega t)$
$y(t) = C_y + r \cdot \sin(\omega t)$

- Ellipse:

$x(t) = C_x + r_x \cdot \cos(\omega t)$
$y(t) = C_y + r_y \cdot \sin(\omega t)$

Where:

- (C_x, C_y) is the center of the orbit.
- r is the radius (or r_x, r_y for an ellipse).
- ω is the angular velocity, controlling how fast it orbits.
- t is the time.

Implementation

This method is effective for a "power throw" that orbits a point in front of the player. The center of the circle/ellipse can be set at a fixed distance from the player at the moment of the throw. The boomerang then travels along this path for a set duration or number of rotations before transitioning to the INBOUND state.

Table 1: Comparison of Outbound Trajectory Modeling Techniques

Technique	Core Principle	Pros	Cons	Best For	Implementation Complexity
Parametric Arc	Simulated gravity and momentum	- Feels physical and intuitive - Emergent interactions with game physics - Simple to implement basic version	- Can feel generic - Harder to use for precise, non-standard paths - Player momentum can make it less predictable	Action-focused combat, direct targeting	Low
Bézier Spline	Designer/player-defined curve	- Complete artistic control over path - Highly predictable and reliable -	- Can feel "on-rails" and less physical - Less emergent behavior - Requires more	Puzzle-platforming, strategic combat, stylized motion	Medium

		Excellent for puzzles and hitting targets around obstacles	complex setup for control points		
Circular/Elliptical	Parametric orbital motion	- Excellent for area denial and crowd control - Creates a unique and visually interesting weapon path - Simple mathematical foundation	- Not suitable for precision targeting - Can be difficult to balance - May feel less interactive than other models	Crowd control, creating temporary hazards, boss fights	Medium

---

Section 4: The Return Path: Implementing Homing Behavior

Once the boomerang transitions to its INBOUND state, its primary objective is to return to the player. Since the player is a moving target, a robust homing algorithm is required. The choice of algorithm will significantly impact the feel of the return phase, turning it from a simple background event into a potential gameplay mechanic in its own right.

4.1 Simple Vector-Based Homing

This is the most direct and computationally inexpensive method for implementing homing behavior.

The Logic

In every frame, the algorithm calculates the straight-line vector from the boomerang's current position to the player's current position. This vector is then normalized (its length is set to 1) to get a pure direction, which is then multiplied by a constant return speed to determine the boomerang's velocity for that frame.

Pseudocode

```
// Pseudocode for Simple Vector Homing
void UpdateInbound(float delta_time) {
    // Calculate the direction vector to the player
    Vector2 direction_to_player = (player.position - boomerang.position);

    // Normalize the vector to get a pure direction
    direction_to_player.normalize();

    // Apply velocity in that direction
    boomerang.velocity = direction_to_player * return_speed;
    boomerang.position += boomerang.velocity * delta_time;
}
```

Analysis

This method is highly effective and guarantees the boomerang will always travel directly towards the player's latest position. However, its primary drawback is that it can feel robotic and unnatural. Because the direction is recalculated and applied instantly each frame, the boomerang will make sharp, instantaneous turns to track a fast-moving player, lacking any sense of weight or momentum.

4.2 Smooth Homing with Interpolation (Lerp)

To achieve a more natural, aesthetically pleasing return path that arcs gracefully towards the player, linear interpolation (Lerp) can be used to smooth out the directional changes.

The Logic

Instead of instantly snapping the boomerang's velocity vector to the target direction, this method gradually rotates the current velocity vector towards the target direction over several frames. The Lerp function is perfect for this, as it calculates an intermediate point between two values.

Pseudocode

```
// Pseudocode for Smooth Homing with Lerp
void UpdateInbound(float delta_time) {
    // 1. Get the ideal target direction
    Vector2 target_direction = (player.position - boomerang.position).normalized();

    // 2. Get the boomerang's current direction of travel
    Vector2 current_direction = boomerang.velocity.normalized();

    // 3. Interpolate between the current and target directions
    // 'turn_speed' is a value (e.g., between 1.0 and 10.0) that controls responsiveness
    Vector2 new_direction = lerp(current_direction, target_direction, turn_speed * delta_time);

    // 4. Apply velocity using the new, smoothed direction
    boomerang.velocity = new_direction.normalized() * return_speed;
    boomerang.position += boomerang.velocity * delta_time;
}
```

Analysis

The turn_speed parameter becomes a powerful tuning variable for game feel. A low turn_speed will result in a wide, sweeping return arc, as the boomerang will have a large "turning radius." This can introduce a new gameplay element where the player must anticipate the arc and position themselves correctly for an efficient catch. A high turn_speed will cause the boomerang to turn more sharply, approaching the behavior of the simple vector method. This technique provides a superior sense of momentum and weight, making the boomerang feel like a physical object rather than a simple homing missile.

4.3 Advanced Homing: Boids Algorithm Concepts

For a truly unique and stylized return, particularly for a weapon with a "magical" or "sentient" quality, concepts can be borrowed from Craig Reynolds' Boids flocking algorithm.[22] This involves simulating multiple steering behaviors that combine to produce a complex and organic final path.

The Logic

Instead of a single homing force, the boomerang's acceleration is calculated as a weighted sum of several forces:

- **Seek (Cohesion):** The primary force that steers the boomerang towards the player's position. This is analogous to the simple vector method.
- **Separation:** A repulsive force that steers the boomerang away from nearby obstacles (e.g., walls or other projectiles). This can help it navigate complex environments without getting stuck.
- **Alignment/Flow Field:** A force that attempts to align the boomerang's velocity with a predefined "flow field" in the level. This can be used to create beautiful, swooping paths that follow the contours of the environment, making the return journey visually spectacular.

Analysis

This is an advanced technique that is computationally more expensive but offers the highest degree of artistic control over the weapon's movement. It is best suited for a game where the boomerang's path is a central visual and mechanical element. The interplay of these forces can lead to highly emergent and impressive behaviors that feel intelligent and alive. The simple, aggressive homing of the first method makes catching the boomerang a trivial part of the process, keeping the player's focus on the throw. In contrast, a smoother, arcing return path created by interpolation or Boids-like forces elevates the return into a gameplay mechanic of its own, requiring the player to actively think about their positioning to retrieve their weapon efficiently.

Section 5: Creating Impact: A Deep Dive into Collision Systems

A projectile is defined not just by its path, but by its interaction with the world. A robust and well-differentiated collision system is what transforms the boomerang from a moving sprite into a functional and satisfying game mechanic. This requires careful hitbox design, a logical system for filtering collision types, and specific, state-dependent responses for every kind of interaction.

5.1 Hitbox Design and Best Practices

The hitbox is the invisible shape that the physics engine uses to determine if a collision has occurred.[6] It is a gameplay tool, and its design should prioritize function over visual fidelity.

- **Shape and Size:** For a spinning boomerang sprite, a perfectly form-fitting hitbox can

be complex and lead to frustrating "near misses" where the visual sprite appears to hit but the hitbox does not. A more generous, simpler shape like a circle or a "pill" (a capsule shape) often provides a better player experience. This shape should represent the weapon's intended area of effect, not its literal pixel-perfect outline.[6]

- **The "Bullet Through Paper" Problem:** This is a critical issue in discrete collision detection systems, where an object's position is checked only once per frame. A fast-moving boomerang can be on one side of a thin wall in one frame and completely on the other side in the next, never registering a collision.[23] To solve this, developers can use one of two primary techniques:
    1. **Continuous Collision Detection (CCD):** Many modern physics engines offer CCD as a built-in option for fast-moving objects. It works by extruding the object's hitbox along its velocity vector to calculate the precise time and point of impact within a frame.
    2. **Manual Raycasting:** If CCD is not available or is too performance-intensive, a simpler solution is to cast a ray (or a shape-cast using the hitbox) from the boomerang's position in the previous frame to its position in the current frame. If this ray intersects with an object, a collision is registered.[24]

5.2 The Power of Collision Layers and Masks

To prevent the boomerang from wastefully checking for collisions against every object in the game and to allow for differentiated responses, a system of physics layers and masks is essential. This is a standard feature in modern game engines that allows for efficient filtering of collision events.[26]

- **Defining Layers:** Objects in the game are assigned to specific layers. A typical setup would be:
    - **Layer 1: WorldGeometry** (for static level elements like walls, floors, and platforms)
    - **Layer 2: Player** (for the player character)
    - **Layer 3: Enemies** (for all enemy characters)
    - **Layer 4: Boomerang** (for the boomerang projectile itself)
    - **Layer 5: Collectibles** (for items the boomerang should pass through)
- **Setting the Mask:** The boomerang's "collision mask" is then configured to define which layers it should detect collisions with. For example, the boomerang's mask would be set to detect layers 1, 2, and 3 (WorldGeometry, Player, Enemies), but it would ignore layers 4 and 5, preventing it from colliding with itself, other projectiles, or collectibles.

5.3 Defining Differentiated Collision Responses

The core of an effective collision system lies in defining unique responses based on what was

hit and what state the boomerang was in at the time of impact.

Enemy Collision

When the boomerang's hitbox overlaps with an object on the Enemies layer:

- **Response:** The primary response is to call a function on the enemy object, such as TakeDamage(damage_amount).
- **State-Dependence:** This response can be state-dependent. For example, the boomerang might only deal damage while in the OUTBOUND or APEX states. In the INBOUND state, it could pass harmlessly through enemies, preventing accidental self-damage if an enemy is close to the player.
- **Piercing vs. Non-Piercing:** The weapon's behavior after impact is a key design choice. A can_pierce boolean can be checked. If false, the boomerang transitions to the INBOUND state after the first hit. If true, it continues on its path. To prevent a single piercing boomerang from hitting the same enemy multiple times in one throw, a list of already-hit enemies should be maintained and checked before applying damage.

Environment Collision (WorldGeometry)

When the boomerang collides with an object on the WorldGeometry layer:

- **Response:** This collision should almost always trigger a state change to INBOUND. However, the nature of this interaction defines the weapon's personality.
    - **Hard Stop & Return:** The most common and predictable behavior. The boomerang's motion stops, and it immediately begins its return journey.
    - **Ricochet/Bounce:** For a more dynamic and chaotic "high-powered" feel, the boomerang's velocity can be reflected off the wall's surface normal. This can turn missed shots into lucky hits and allows for high-skill trick shots, fundamentally changing the weapon's strategic use.
    - **Destruction:** In some cases, the boomerang might simply be destroyed on impact with a wall.

Player Collision (Player)

When the boomerang collides with an object on the Player layer:

- **Response:** This collision should only be actively checked for during the INBOUND state.
- **Action:** The collision triggers a transition back to the HELD state. The boomerang's game object is deactivated (or returned to an object pool for performance), and the player's ability to throw again is enabled, completing the gameplay loop.

The strategic niche of the weapon is defined by these responses. The Rolling Cutter from *Mega Man*, for instance, famously passes through walls and enemies, establishing its role as

an unstoppable tool for screen control.[27] In contrast, the boomerang in *The Legend of Zelda: A Link to the Past* typically returns immediately upon hitting a solid object, reinforcing its role as a utility item for stunning single targets or retrieving items.[29] By carefully crafting these rules of interaction, a developer can guide the player toward using the weapon in creative and intended ways.

Section 6: Polishing the Mechanic: Achieving a "High-Powered" Feel

A mechanically functional weapon is only half the battle. The perception of "power" is an illusion crafted through the careful layering of audio-visual feedback. This "juice" is what elevates a system of code and numbers into a satisfying, visceral experience. It is not mere polish to be added at the end of development; it is an integral part of the mechanic's design from the outset.

6.1 Visual Feedback: The Language of Power

Visual cues provide the player with immediate, subconscious information about the boomerang's state and impact.

- **Sprite Animation:** The boomerang's sprite should not be static. It must rotate to sell the fantasy of a spinning projectile. The speed of this rotation can be linked to its velocity, spinning faster when thrown and slowing as it reaches its apex, providing a subtle but effective physical cue.
- **Particle Effects:** Judicious use of particles can dramatically enhance the feeling of power.
    - **Trail Renderer:** A trail of light or energy that follows the boomerang is the most effective way to communicate its path and speed. The color, width, or intensity of this trail can change depending on its state. For example, it might be a sharp, energetic color in the OUTBOUND state and a softer, wispier color in the INBOUND state.
    - **Impact Sparks:** A bright, explosive particle burst upon hitting an enemy provides a satisfying confirmation of a successful hit. A different effect, like a shower of metallic sparks, should be used for wall collisions to clearly differentiate the two outcomes.
    - **Charge-Up Effect:** If the throw can be charged for more power, this must be communicated with a growing visual effect on the player character or the held boomerang itself, signaling to the player that their input is having an effect.
- **Screen Shake and Hit Stop:** These are two of the most potent tools for making impacts feel powerful.

    - **Screen Shake:** A very slight, brief shake of the camera upon enemy impact can make the hit feel momentous.
    - **Hit Stop:** Freezing the game for a minuscule duration (1-3 frames) at the moment of impact creates a sense of immense force. This technique is used extensively in action games to add weight and crunch to attacks.

6.2 Audio Feedback: The Sound of Satisfaction

Sound design is arguably even more important than visuals for creating a satisfying gameplay loop. Each phase of the boomerang's use should have a distinct and well-designed sound effect.

- **The Throw:** A sharp, impactful sound—a "shing" or a powerful "thwump"—that communicates the initial force of the throw.
- **The Flight Loop:** A continuous "whoosh" or "hum" that plays while the boomerang is in the air. The pitch and volume of this sound should be modulated by the boomerang's velocity. As it flies away at high speed, the pitch is high; as it slows at the apex, the pitch drops; as it accelerates back toward the player, the pitch rises again. This provides a constant, intuitive audio stream of information about the weapon's state.
- **The Impact:** A meaty, "crunchy" sound for hitting an enemy is essential for satisfaction. A sharp, metallic "clank" for hitting a wall provides clear, unambiguous feedback.
- **The Catch:** A clean, crisp sound effect—a "thwip," "click," or "sheathe" sound—when the player successfully catches the boomerang. This sound provides closure to the gameplay loop and signals to the player that the weapon is ready to be used again. The absence of this sound after a throw is a clear indicator that the weapon is still in flight.

6.3 Case Study Synthesis: Learning from the Masters

Examining classic boomerang-like weapons reveals how these principles are combined to create iconic and memorable mechanics.

- ***Mega Man*'s Rolling Cutter:** This weapon's power comes from its reliability and screen control. Its unique teardrop-shaped arc and ability to pass through walls make it a predictable tool for clearing out enemies above and below the player's direct line of fire.[27] Its visual design is simple—a spinning blade—but its function is what makes it powerful.
- ***The Legend of Zelda: A Link to the Past*'s Magical Boomerang:** The upgraded red boomerang flies faster and farther than its predecessor, but its primary function remains utility.[29] It is used to stun most enemies, temporarily paralyzing them to create an opening for a sword attack.[30] Its power is not in its own damage, but in how it enables the player's primary damage source.
- ***Castlevania: Symphony of the Night*'s Chakram:** This weapon prioritizes speed and

player mobility. It is a fast, straight-line projectile that hits multiple times and, crucially, does not lock Alucard into an attack animation, allowing him to keep moving while attacking.[31] Its power is derived from its high damage-per-second potential and the tactical freedom it grants the player.

Each of these classic weapons demonstrates that "power" is a multifaceted concept. It can mean raw damage, tactical utility, screen control, or player mobility. The feeling of power is a carefully constructed illusion, born from the synergy of the underlying mechanics and the audio-visual feedback that communicates their impact to the player. A mechanically weak weapon can be made to feel powerful with strong feedback, while a statistically strong weapon will feel lackluster if its impacts are silent and invisible. The developer must treat these feedback systems as a core component of the weapon's design, not as an afterthought.

Section 7: An Alternative Architecture: The Entity Component System (ECS)

The Entity Component System (ECS) is a software architectural pattern, popular in modern game development, that offers a powerful alternative to traditional Object-Oriented Programming (OOP).[33] Instead of creating a monolithic
Boomerang class that contains all its data and logic, ECS follows the principle of composition over inheritance by separating data from behavior.[33] This approach can lead to more flexible, reusable, and performant code.[35]

7.1 Deconstructing the Object: The Core Concepts of ECS

In an ECS architecture, a game object is broken down into three fundamental parts [34]:

- **Entity:** An entity is not an object in the traditional sense. It is simply a unique identifier, like a number, that represents a single "thing" in the game world, such as our boomerang.[33] It contains no data or logic itself.
- **Component:** A component is a plain data structure (or "struct") that holds a specific piece of data for an entity.[34] Components have no behavior. For a boomerang, you might have a
  TransformComponent for its position, a VelocityComponent for its movement, and a BoomerangPropertiesComponent for its unique attributes like range and return speed.
- **System:** A system contains all the logic. Systems operate on collections of entities that possess a specific set of components.[33] For example, a
  MovementSystem would query for all entities that have both a TransformComponent and a VelocityComponent and update their positions every frame.

This separation avoids the rigid hierarchies of OOP, where a "Goblin Shopkeeper" might create a confusing inheritance problem.[36] In ECS, you would simply create an entity and attach

GoblinComponent, ShopkeeperComponent, and TransformComponent to it, allowing for maximum flexibility.[35] Furthermore, by grouping all instances of a single component type together in memory, systems can process them with high efficiency, which is a significant performance advantage.[33]

7.2 The Boomerang as an ECS Entity

To build our boomerang in an ECS framework, we would first create an entity. This entity would then be defined by the collection of components attached to it.

- **The Boomerang Entity:** A unique ID, for instance boomerang_01.
- **Associated Components (Data):**
    - TransformComponent: Stores its position and rotation.
    - VelocityComponent: Stores its current linear_velocity and angular_velocity.
    - BoomerangPropertiesComponent: Contains boomerang-specific data like max_range, outbound_duration, return_speed, and time_in_flight.
    - CollisionComponent: Defines its hitbox_shape, size, and collision_mask.
    - RenderableComponent: Holds information for drawing, such as a sprite_id or trail_effect_id.
- **Governing Systems (Logic):**
    - MovementSystem: Queries for all entities with TransformComponent and VelocityComponent. Each frame, it updates the position based on the linear_velocity.
    - BoomerangControlSystem: This is the core logic system for the weapon. It queries for entities that have TransformComponent, VelocityComponent, and BoomerangPropertiesComponent. This system would be responsible for implementing the state-based behaviors (outbound, inbound, etc.).
    - CollisionSystem: Queries for entities with TransformComponent and CollisionComponent. It checks for overlaps with other entities and triggers events or modifies components accordingly (e.g., telling the BoomerangControlSystem to initiate a return state).
    - RenderSystem: Queries for entities with TransformComponent and RenderableComponent and handles drawing them to the screen.

7.3 Implementing the State Machine in ECS

The Finite State Machine (FSM) described in Section 2 is not replaced by ECS; rather, ECS provides a robust and data-oriented way to implement it. There are two primary strategies for managing states within an ECS architecture.[38]

Method 1: State as Tag Components

In this approach, each state is represented by an empty component, often called a "tag." To define the boomerang's state, you would add or remove these tags.[40]

- **State Components:** HeldStateTag, OutboundStateTag, ApexStateTag, InboundStateTag.
- **Logic:** An entity's current state is defined by which tag it possesses. For example, a boomerang currently flying away from the player would have the OutboundStateTag.
- **Transitions:** A state transition is a "structural change": the BoomerangControlSystem removes the OutboundStateTag and adds the InboundStateTag when the return condition is met.
- **Benefit:** This allows for highly specialized systems. For instance, a HomingSystem could be written to query *only* for entities that have an InboundStateTag. This keeps systems small, focused, and highly efficient, as they only ever see the entities they care about.[41]

Method 2: State as a Value in a Component

This method uses a single component to hold the current state, typically as an enum.[38]

- **State Component:** StateComponent { enum CurrentState { HELD, OUTBOUND, APEX, INBOUND } }.
- **Logic:** A single, more complex BoomerangControlSystem queries for all boomerang entities and uses an internal switch statement to execute the correct logic based on the CurrentState value.
- **Transitions:** A state transition is a simple data modification (e.g., state_component.CurrentState = INBOUND).
- **Benefit:** This avoids structural changes (adding/removing components), which can be more performant if many entities are changing state every frame.[40] It keeps all of the state logic consolidated in one place.
- **Drawback:** The system can become large and complex. It may also lead to less efficient data access, as the system has to fetch all boomerang entities, including those in states it might not act upon in a given frame.[40]

By adopting an ECS architecture, the boomerang's implementation becomes inherently modular and scalable. Adding a new behavior, such as a "sticky" boomerang, is as simple as creating a StuckOnWallStateTag and a corresponding system to manage its logic, without altering any of the existing systems. This data-centric approach provides a powerful and modern framework for building complex and dynamic game mechanics.

Conclusion

The development of a high-powered boomerang weapon for a 2D platformer is a microcosm of the game design process itself. It demands a thoughtful balance between technical implementation and artistic vision, between mathematical precision and the intangible quality of "game feel." The journey from concept to a fully realized, satisfying mechanic relies on a series of deliberate design choices, not a quest for physical realism.

Adopting an Entity Component System (ECS) architecture provides a flexible and data-oriented foundation for this process. The boomerang ceases to be a single, monolithic object and instead becomes an Entity defined by its Components—small, reusable packets of data like TransformComponent, VelocityComponent, and BoomerangPropertiesComponent. The logic for its movement, whether a predictable path-defined trajectory or a dynamic force-based one, is handled by dedicated Systems that operate on entities possessing the correct combination of components.

The weapon's complex, multi-phase behavior is managed by implementing a state machine within the ECS framework. Instead of traditional states, the boomerang's current phase is determined by the presence of "tag" components like OutboundStateTag or InboundStateTag. A state transition is simply the act of removing one tag and adding another, which in turn dictates which systems—such as an OutboundPathSystem or a HomingSystem—will process the entity. The mathematics of the path, whether a parametric arc or a Bézier curve, become pure data within a component, interpreted by the appropriate system to update the entity's position.

Ultimately, the mechanical systems are brought to life through a rich layer of audio-visual feedback, also managed through the ECS. A CollisionSystem can detect an impact and add an ImpactEffectComponent to an entity, which is then picked up by an AudioSystem and a RenderSystem to play the correct sound and spawn particles. By composing behavior from data and logic from discrete systems, a developer can craft a boomerang weapon that is not only functional and performant but also modular, extensible, and deeply satisfying to wield.

Works cited

1. Boomerang Theory, accessed August 1, 2025, https://www3.eng.cam.ac.uk/~hemh1/boomerang_theory.pdf
2. Boomerang Flight Dynamics, accessed August 1, 2025, https://iypt.ru/wp-content/uploads/2024/10/Boomerang-Flight-Dynamics.pdf
3. What Makes Boomerangs Come Back?, accessed August 1, 2025, https://www.math.uci.edu/~eesser/papers/justboom.pdf
4. Boomerang Flight Tests - Aerospace Research Central, accessed August 1, 2025, https://arc.aiaa.org/doi/pdfplus/10.2514/6.2014-3127
5. Boomerang simulator game, accessed August 1, 2025, https://decoboomerangs.com/en/resource/online-simulator
6. accessed December 31, 1969,
7. How to create a boomerang-a-like path? - Game Development Stack Exchange, accessed August 1, 2025, https://gamedev.stackexchange.com/questions/150379/how-to-create-a-boomerang-a-like-path

8. Curved Paths - Red Blob Games, accessed August 1, 2025, https://www.redblobgames.com/articles/curved-paths/
9. Sgt. Boomerang Devblog-Part 1 – nevzatarman, accessed August 1, 2025, https://nevzatarman.com/2020/06/01/sgt-boomerang-devblog-part-1/
10. trying to make a boomerang mechanic : r/gamemaker - Reddit, accessed August 1, 2025, https://www.reddit.com/r/gamemaker/comments/1hhlxfk/trying_to_make_a_boomerang_mechanic/
11. State Machine Setup for 2D Platformer Character ~ Godot 4 GameDev Tutorial - YouTube, accessed August 1, 2025, https://www.youtube.com/watch?v=fuGiJdMrCAk
12. Boomerang2D - Storm Garden Studio, accessed August 1, 2025, https://www.stormgardenstudio.com/Boomerang2D
13. 2D Projectile tracing path clarification - Stack Overflow, accessed August 1, 2025, https://stackoverflow.com/questions/10935060/2d-projectile-tracing-path-clarification
14. 2D projectile motion (article) | Khan Academy, accessed August 1, 2025, https://www.khanacademy.org/science/in-in-class11th-physics/in-in-class11th-physics-motion-in-a-plane/in-in-class11-two-dimensional-projectile-motion/a/what-is-2d-projectile-motion
15. 4.4: Projectile Motion - Physics LibreTexts, accessed August 1, 2025, https://phys.libretexts.org/Bookshelves/University_Physics/University_Physics_(OpenStax)/Book%3A_University_Physics_I_-_Mechanics_Sound_Oscillations_and_Waves_(OpenStax)/04%3A_Motion_in_Two_and_Three_Dimensions/4.04%3A_Projectile_Motion
16. 4.4 Projectile Motion – Biomechanics of Human Movement, accessed August 1, 2025, https://pressbooks.bccampus.ca/humanbiomechanics/chapter/3-4-projectile-motion-2/
17. Let's discuss 2d platforming jumps : r/gamedev - Reddit, accessed August 1, 2025, https://www.reddit.com/r/gamedev/comments/3mjjrf/lets_discuss_2d_platforming_jumps/
18. Bezier curves are cool, we use them for our throwing mechanic : r/godot - Reddit, accessed August 1, 2025, https://www.reddit.com/r/godot/comments/q1a2sa/bezier_curves_are_cool_we_use_them_for_our/
19. gamedev.stackexchange.com, accessed August 1, 2025, https://gamedev.stackexchange.com/questions/9607/moving-an-object-in-a-circular-path#:~:text=X%20%3A%3D%20originX%20%2B%20cos(angle,That's%20it.
20. Moving an object in a circular path - Game Development Stack Exchange, accessed August 1, 2025, https://gamedev.stackexchange.com/questions/9607/moving-an-object-in-a-circular-path
21. Parametric Equation of an Ellipse - Math Open Reference, accessed August 1, 2025, https://www.mathopenref.com/coordparamellipse.html

22. Code Things To Move Like Birds | The Flocking Algorithm In GML - YouTube, accessed August 1, 2025, https://www.youtube.com/watch?v=dkws3E2AT34
23. 2D Platform Games Part 1: Collision Detection for Dummies | Katy's Code, accessed August 1, 2025, https://katyscode.wordpress.com/2013/01/18/2d-platform-games-collision-detection-for-dummies/
24. How do you handle 2D projectile collision detection? : r/gamedev - Reddit, accessed August 1, 2025, https://www.reddit.com/r/gamedev/comments/4tj7be/how_do_you_handle_2d_projectile_collision/
25. In a game program, do bullets check if they hit an enemy or do enemies check if they are hit by a bullet? : r/gamedev - Reddit, accessed August 1, 2025, https://www.reddit.com/r/gamedev/comments/1inoeys/in_a_game_program_do_bullets_check_if_they_hit_an/
26. I'm trying to allow enemies to pass through walls - godot - Reddit, accessed August 1, 2025, https://www.reddit.com/r/godot/comments/ymjwnw/im_trying_to_allow_enemies_to_pass_through_walls/
27. Get Equipped Analysis: Rolling Cutter - The Mega Man Network, accessed August 1, 2025, https://themmnetwork.com/2014/04/28/get-equipped-analysis-rolling-cutter/
28. Rolling Cutter - MegaManMaker Wiki, accessed August 1, 2025, https://wiki.megamanmaker.com/index.php/Rolling_Cutter
29. Magical Boomerang - The Legend of Zelda: A Link to the Past Guide ..., accessed August 1, 2025, https://www.ign.com/wikis/the-legend-of-zelda-a-link-to-the-past/Magical_Boomerang
30. [TLoZ] [LA] [WW] [ALTTP] I know we get a Boomerang in multiple titles, but do we really use it as intended? : r/truezelda - Reddit, accessed August 1, 2025, https://www.reddit.com/r/truezelda/comments/13mrrvs/tloz_la_ww_alttp_i_know_we_get_a_boomerang_in/
31. Looking for a weapon in SOTN : r/castlevania - Reddit, accessed August 1, 2025, https://www.reddit.com/r/castlevania/comments/19ak9ez/looking_for_a_weapon_in_sotn/
32. Rare drops in the first castle - Castlevania: Symphony of the Night - GameFAQs, accessed August 1, 2025, https://gamefaqs.gamespot.com/boards/196885-castlevania-symphony-of-the-night/77263480?page=1
33. Entity component system - Wikipedia, accessed August 1, 2025, https://en.wikipedia.org/wiki/Entity_component_system
34. Entity Component System: An Introductory Guide - Simplilearn.com, accessed August 1, 2025, https://www.simplilearn.com/entity-component-system-introductory-guide-article
35. All about Entity Component System - Community Tutorials - Developer Forum |

Roblox, accessed August 1, 2025, https://devforum.roblox.com/t/all-about-entity-component-system/1664447
36. A Simple Entity Component System (ECS) [C++] - Austin Morlan, accessed August 1, 2025, https://austinmorlan.com/posts/entity_component_system/
37. OOP vs ECS - is my understanding correct? : r/gamedev - Reddit, accessed August 1, 2025, https://www.reddit.com/r/gamedev/comments/1i19yla/oop_vs_ecs_is_my_understanding_correct/
38. ECS and finite state machine · skypjack entt · Discussion #1230 - GitHub, accessed August 1, 2025, https://github.com/skypjack/entt/discussions/1230
39. oop - Finite State Machine Implementation in an Entity Component ..., accessed August 1, 2025, https://stackoverflow.com/questions/39185133/finite-state-machine-implementation-in-an-entity-component-system
40. Implement state machines | Entities | 1.3.14 - Unity - Manual, accessed August 1, 2025, https://docs.unity3d.com/Packages/com.unity.entities@1.3/manual/state-machine.html
41. Finite State Machines with Ash entity component system framework, accessed August 1, 2025, https://www.richardlord.net/blog/ecs/finite-state-machines-with-ash