The Axiomatic Engine: A Comprehensive Architectural Proposal for Integrating Non-Euclidean Stochastics and Mathematical Metaphysics into the Tarmin Framework

1. Executive Summary and Theoretical Framework

The development trajectory of *Tarmin2Revised*, as currently delineated in the project's design documentation, focuses on a high-fidelity recreation of classic dungeon crawler mechanics, augmented by modern systems such as dynamic weather [1], biome-specific fog of war [1], and a sophisticated gore rendering system.[1] While these enhancements successfully modernize the user experience, they operate within the traditional paradigms of the Role-Playing Game (RPG) genre: abstract random number generation (RNG), linear statistical progression, and static combat states. This research report proposes a fundamental divergence from this trajectory, introducing a meta-layer of gameplay defined as **The Axiomatic Engine**.

The core proposition of this architectural overhaul is to replace the abstract random number generators—specifically the standard java.util.Random instances currently utilized in CombatManager.java [1]—with a deterministic, physics-based simulation of "chance." This is not merely an aesthetic addition of 3D dice; it is a ludological shift from *randomness* to *chaos*. Randomness is a computed value, opaque to the player; chaos is a physical process, sensitive to initial conditions (input vector, friction, mass), which the player can observe, influence, and master. To execute this, we propose integrating a high-fidelity rigid body dynamics simulation (via the Bullet Physics wrapper for LibGDX) that utilizes "impossible" geometries derived from advanced mathematics—such as Calabi-Yau manifolds and projections of the Fischer-Griess Monster Group—as the physical agents of combat resolution.

Furthermore, this mechanical shift serves as the foundation for a narrative and systemic overhaul grounded in mathematical metaphysics. We propose a "Descent to Simplicity"

progression curve. Contrary to standard RPGs where complexity increases with level, the player in this framework begins in a state of "Eldritch Abstraction," surrounded by chaotic, high-level mathematical concepts (Topology, Set Theory) that produce unpredictable gameplay effects. As the player progresses through the MazeChunkGenerator levels [1], they "decode" the universe, discovering foundational "Axioms"—algebraic and arithmetic constants (π, e, 1, 0)—which grant supreme deterministic power. This thematic inversion appeals directly to the target demographic's intellectual curiosity ("math nerd culture") while providing a coherent explanation for the game's escalating power scale.

This document serves as an exhaustive implementation guide for this overhaul, detailing the necessary modifications to the existing codebase (including Tarmin2.java, CombatManager.java, and FirstPersonRenderer.java), the integration of FastNoiseLite [1] for procedural mesh generation of the dice, and the expansion of the ItemModifier system [1] to support equation-based synergy.

---

2. The Stochastic Combat Engine: Physics as Gameplay

The current combat architecture, as defined in CombatManager.java [1], relies on a straightforward state check. The method checkForAdjacentMonsters() identifies a target, and if the player initiates an attack, the damage is calculated instantly based on stat comparisons. This efficiency, while functional, lacks the tactile engagement and "gambling" addictiveness requested. The proposed **Stochastic Combat Engine** interrupts this instantaneous loop, inserting a physics simulation phase that demands player agency and resource wagering.

2.1. From Abstract RNG to Deterministic Chaos

The shift from Random.nextInt() to a physics simulation fundamentally alters the nature of "critical hits" and "misses." In a standard RNG system, a critical hit is a boolean state triggered by a threshold (e.g., roll > 0.95). In a physics system, a critical hit becomes a geometric rarity—a specific face of a 196,883-dimensional object landing perpendicular to the gravity vector.

2.1.1. Architectural Integration of the Physics World

The project is built on LibGDX.[1] To achieve the requested "incredibly accurate physics," we must integrate the **Bullet Physics** library (gdx-bullet). This requires significant changes to the main game loop in Tarmin2.java [1] and the rendering pipeline in GameScreen.java.[1]

The existing FirstPersonRenderer [1] utilizes a custom raycasting engine to draw the dungeon walls in vertical strips (evident from the drawWallStrip logic and depth buffer usage). This 2.5D rendering technique cannot natively display true 3D objects like rolling dice with rotational physics. Therefore, the physics simulation must exist in a separate coordinate space, rendered via a secondary PerspectiveCamera overlay.

Implementation Strategy:
The integration requires a layered rendering approach within the render() method of GameScreen.java 1:

1. **Layer 1: The Raycaster:** The FirstPersonRenderer draws the maze, monsters, and gore decals.[1] This creates the background context.
2. **Layer 2: The Atmospheric/Weather Layer:** The WeatherRenderer [1] draws rain, fog, or snow on top of the maze. This is crucial for the "Physics Synergy" discussed later (e.g., rain reducing the friction of the dice tray).
3. **Layer 3: The Stochastic Overlay (New):** This is a new ModelBatch rendering the 3D dice and the "Tray" environment. This layer must have a transparent background to allow the dungeon to remain visible, maintaining immersion.
4. **Layer 4: The HUD:** Finally, the Hud [1] draws the UI elements on top.

The CombatManager [1] must be refactored to include a new state: CombatState.PHYSICS_RESOLUTION. When a player initiates an attack, the manager does not immediately call monster.takeDamage(). Instead, it transitions to PHYSICS_RESOLUTION, instantiates the btDiscreteDynamicsWorld, and spawns the rigid bodies corresponding to the player's equipped dice. Only when the physics bodies fall below a velocity threshold (the "sleeping" state) does the manager read the result and apply damage.

2.1.2. Simulation Parameters and Player Influence

The "influence from player build" requested in the query is achieved by mapping PlayerStats [1] to the physical constants of the simulation.

- **War Strength (W) $\rightarrow$ Impulse Force:** The raw force vector applied to the dice throw is proportional to the player's War Strength. A high War Strength allows for

"violent" rolls that can knock heavy "Obstacle Pins" out of the tray (a mechanic to lower enemy armor).
- **Spiritual Strength (S) $\rightarrow$ Friction Coefficients:** Spiritual Strength represents control over reality. In the physics engine, this maps to the friction of the tray surface. High Spirit lowers friction, allowing dice to slide further and potentially traverse "Multiplier Zones" painted onto the tray texture.
- **Dexterity (Proposed Stat) $\rightarrow$ Torque:** While dexterity is currently implicit, explicit control over the *spin* (torque) applied to the dice allows skilled players to bias the roll towards specific axes, mimicking the technique of "dice setting."

2.2. Impossible Geometries: The Artifacts of Chance

The user query calls for "crazy variations" of dice based on high-level math concepts. Standard Platonic solids (d4, d6, d8, d12, d20) are insufficient. We must procedurally generate meshes representing higher-dimensional objects projected into 3D space.

2.2.1. The Calabi-Yau Die (The Topological Roller)

Calabi-Yau manifolds are complex shapes used in string theory to compactify dimensions. They are characterized by holes, ridges, and non-trivial topology.

- **Mesh Generation:** We can utilize the FastNoiseLite library [1] already present in the project. By mapping 3D noise to the vertices of a subdivided sphere and using the GetNoise(x, y, z) function to displace vertices negatively, we can create deep concavities that mimic the "holes" of a Calabi-Yau manifold.
- **Physics Interaction:** Standard convex hull collision (btConvexHullShape) is unsuitable here because it "shrink-wraps" the object, covering the holes. We must use btGImpactMeshShape, which supports concave collision.
- **Gameplay Mechanic:** If the die lands on a "hole" (a region where the center of mass is unsupported), the physics engine will struggle to settle it. We can detect this "unstable equilibrium." If the die balances precariously, it triggers a "Dimensional Rift," dealing damage to *both* the player and the monster (Chaos Damage).

2.2.2. The Monster Group Die (The Symmetry Breaker)

The Fischer-Griess Monster Group is the largest sporadic simple group. A physical representation would be a sphere faceted with an absurd number of planes.

- **Mesh Generation:** A geodesic sphere with a subdivision level of 6 or 7, resulting in thousands of micro-faces.
- **Material Properties:** This die is assigned a material with extremely high restitution (bounciness) via btMaterial. It behaves almost like a superball.
- **Resolution Mechanic:** Because the faces are too small to have numbers, the "result" is determined by the **normal vector** of the face pointing up relative to the world Y-axis. This vector is compared against a lookup table of "Elemental Vectors."
    - Vector aligns with $(1, 0, 0)$: **Fire Damage** (derived from DamageType.FIRE [1]).
    - Vector aligns with $(0, 0, 1)$: **Void Damage**.
    - Vector aligns with $(0, 1, 0)$: Physical Damage.
      This makes the Monster Die a tool for seeking specific types of damage rather than specific amounts.

2.2.3. The Hypercube (Tesseract) Die

A 4D cube projected into 3D space involves an "inner" cube and an "outer" cube connected by vertices.

- **Visuals:** We use a vertex shader (modifying the existing shaders [1]) to pulsate the mesh, making the inner cube expand through the outer cube, simulating 4D rotation.
- **Physics:** The collision shape remains a rigid compound shape of 8 cubes.
- **Gameplay:** The Tesseract can "phase." If the simulation detects a collision with a "Wall Pin" in the tray, the Tesseract has a probability (based on player Intelligence) to disable collision for 1 frame, passing *through* the wall. This represents "Piercing" damage that ignores the monster's Armor stat.[1]

2.3. The Gambling System: Wagering Survival

To instill the "addictive gameplay" mentioned, the dice roll must not be free. It requires an investment of the game's scarcity metrics: Food, Arrows (as ammunition), and Health. The Player class [1] tracks these integers.

2.3.1. The Wager Interface

Before the physics simulation begins, the "Dice Tray" overlay presents wagering options.

- **"Push" (Cost: 5 Food):** Adds 10% mass to the dice. Heavier dice transfer more kinetic energy to the target. In the physics engine, this is a direct modification of btRigidBody.setMassProps().
- **"Focus" (Cost: 20 Spiritual Strength):** Slows down the simulation time scale (dynamicsWorld.stepSimulation(Gdx.graphics.getDeltaTime() * 0.5f)). This gives the player more time to apply "nudge" forces to the rolling dice.
- **"Blood Price" (Cost: 10 HP):** The player sacrifices health to imbue the die with Gore properties. The die leaves a trail of BloodParticle instances [1] on the tray. If these particles touch an "Enemy Pin," they dissolve it. This links the combat system directly to the "Project Viscera" gore system.[1]

---

3. Mathematical Metaphysics: The Descent to Simplicity

The second pillar of this research is the integration of a meta-narrative that recontextualizes the game world as a degrading mathematical simulation. This addresses the user's desire to "weave a discovery that the game is based on Mathematics." The progression system is inverted: complexity is the starting point (and the problem), while simplicity is the goal (and the ultimate power).

3.1. The Hierarchy of Concepts

We categorize the game's progression into four mathematical epochs, scaling from high-level abstraction down to foundational axioms. This structure dictates item generation, monster behavior, and visual presentation.

Phase 1: The Topological Chaos (Levels 1-4)

- **Concept:** High-level, "eldritch" math. Non-Euclidean geometry, Topology, Knot Theory.
- **Theme:** Unpredictability and Transformation.
- **Data Representation:** Items in LootTable.java [1] at this tier generate with modifiers like MOD_TOPOLOGY.
- **Gameplay Effect:** A "Klein Bottle Potion" has infinite volume but no inside; drinking it might heal the player or invert their stomach (damage). A "Möbius Sword" has only one side; it hits every second turn, but deals double damage (as it travels twice the distance to return to the start).
- **Visuals:** The WallDecal system [1] renders textures that appear to twist and fail to tile correctly, inducing a sense of unease.

Phase 2: The Calculus of Change (Levels 5-9)

- **Concept:** Analysis, Limits, Derivatives, Integrals.
- **Theme:** Rate of change and accumulation over time.
- **Gameplay Effect:** Items operate on variables of time (t) and velocity (v).
    - **The Derivative Dagger ($f'(x)$):** Damage is calculated based on the *change* in the player's position relative to the last frame. The faster the player moves (kiting), the higher the damage.
    - **The Integral Plate ($\int$):** Armor value is the area under the curve of incoming damage over the last 10 turns. If the player takes a massive burst of damage, the armor hardens significantly for the subsequent turns.
- **Code Integration:** This requires tracking historical data in PlayerStats [1], creating a buffer of "Damage Taken" values to compute the integral.

Phase 3: The Algebraic Balance (Levels 10-13)

- **Concept:** Algebra, Geometry, Polynomials.
- **Theme:** Symmetry and Equation Solving.
- **Gameplay Effect:** Abilities resolve variables (x).
    - **Quadratic Maul:** $Damage = ax^2 + bx + c$. The damage curve is parabolic. At close range ($x=1$), damage is moderate. At mid-range ($x=5$), damage peaks. At long range, it drops. This forces precise tactical positioning in the grid-based Maze.[1]
    - **Pythagorean Bow:** If the player, the target, and a wall form a right-angled triangle, the shot deals critical damage ($a^2 + b^2 = c^2$). This utilizes the Pathfinder [1] to

calculate geometric relations on the grid.

Phase 4: The Axiomatic Core (Levels 14+)

- **Concept:** Number Theory, Constants, Arithmetic.
- **Theme:** Absolute, undeniable truth.
- **Gameplay Effect:** Reality breaking.
  - **The Unit Sword (1):** Deals exactly 100% of the enemy's health. It is the concept of "One Hit."
  - **The Null Shield (0):** Incoming damage is multiplied by 0.
  - **The Constant π:** Infinite range, infinite duration.
- **Balance:** These items come with "Universal Constraints." The Null Shield might reduce the player's movement speed to 0, turning them into an immovable singularity.

3.2. Visualizing the Equations

The user requests "showing the bizarre math equations on the screen." This is a UI challenge that must be solved within the Hud class [1] and the shader pipeline.

3.2.1. The Equation Synergy Engine

When items with mathematical modifiers are equipped, the game constructs an **Equation String** to display. We expand ItemModifier [1] to include a String formula field.

- **Example Synergy:**
  - **Weapon:** *Sword of the Tree* (Modifier: Recursive Growth $O(2^n)$).
  - **Ring:** *Band of Limits* (Modifier: $\lim_{n \to \infty}$).
  - **Synergy:** The game detects the interaction between "Growth" and "Limit."
  - Visual: The HUD draws a glowing line connecting the weapon slot and ring slot. Above the health bar, the text renders:

    $$Damage = \lim_{n \to \infty} 2^n = \infty$$
  - **Effect:** The player's next attack deals infinite damage, but destroys the weapon (the limit was reached, reality broke).

3.2.2. Shader-Based Math Visualization

To make the math feel "magical," we use the crt.frag shader.[1] When a high-level equation resolves, we pass a uniform u_mathIntensity to the shader. This triggers a "glitch" effect where the scanlines deform into sine waves, and the color channels separate (chromatic aberration), representing the instability of the simulation.

Additionally, floating damage numbers (typically handled in EntityRenderer [1]) are replaced by the simplification steps.

- **Standard RPG:** "Hit! 50 Damage."
- **Tarmin Axiomatic:**
  - Step 1: $5x^2 + 10$ appears.
  - Step 2: $5(2)^2 + 10$ (substituting the dice roll).
  - Step 3: $20 + 10$.
  - Step 4: **30**.

---

4. Implementation Details: The Code Architecture

This section outlines the specific code modifications required to realize the Axiomatic Engine, referencing the provided source files.

4.1. Expanding the Data Structures

The ModifierType.java enum [1] is the backbone of the item system. It must be expanded to support the new mathematical paradigms.

Current State:

Java

```
public enum ModifierType {
    BONUS_DAMAGE, BONUS_DEFENSE, ADD_FIRE_DAMAGE,...
}
```

Proposed Expansion:

Java

```
public enum ModifierType {
    // Existing
    BONUS_DAMAGE, BONUS_DEFENSE, ADD_FIRE_DAMAGE,

    // Tier 1: Topology
    NON_EUCLIDEAN_HIT, // Chance to bypass armor via 4th dimension
    MOBIUS_STRIKE,     // Double damage, half attack speed

    // Tier 2: Calculus
    DERIVATIVE_SCALING, // Damage scales with player velocity
    INTEGRAL_DEFENSE,   // Defense scales with total damage taken

    // Tier 3: Algebra
    QUADRATIC_RANGE,    // Damage curve based on distance

    // Tier 4: Constants
    CONSTANT_PI,        // Infinite range/mass
    CONSTANT_EULER,     // Exponential growth
    AXIOM_NULL          // Zero interaction (Invincibility/Inaction)
}
```

Similarly, LootTable.java [1] must be updated. The ModInfo inner class currently stores minBonus and maxBonus. This needs to store an EquationTemplate or a reference to a MathFunction interface that can accept game state variables (player speed, enemy health, turn count) as inputs.

4.2. Refactoring CombatManager.java

The CombatManager [1] is the nexus of this overhaul. We must implement a rigorous state machine to handle the physics interruption.

Algorithm 1: The Axiomatic Combat Loop

1. **Trigger:** Player input calls combatManager.playerAttack().
2. **State Transition:** Set CombatState to PRE_ROLL_WAGERING.
3. **UI Overlay:** GameScreen renders the Wagering UI (Bet Food/Arrows).
4. **Input:** Player confirms wager. State transitions to PHYSICS_SIMULATION.
5. **Physics Init:**
    - Retrieve PlayerStats.[1]
    - Calculate Impulse = WarStrength * WagerMultiplier.
    - Calculate TrayFriction = 1.0 / SpiritualStrength.
    - Spawn Dice meshes via BulletPhysicsManager (new class).
6. **Simulation Loop:**
    - Step physics world.
    - Check for "Singularities" (dice balancing on edges).
    - Apply "Tilt" forces from player input (Dexterity check).
7. **Settling:** When totalKineticEnergy < THRESHOLD, State transitions to RESOLUTION.
8. **Math Resolution:**
    - Read top face of dice.
    - Feed result into ItemModifier equations.
    - Calculate final Damage and SideEffects.
9. **Application:** Call monster.takeDamage(). State returns to IDLE.

4.3. Procedural Mesh Generation with FastNoiseLite

The snippets include FastNoiseLite.java.[1] We can leverage this to procedurally generate the "Impossible" dice meshes at runtime, ensuring no two "Chaos Dice" are identical.

Implementation Logic:
To create a "Monster Group" style die (a rough, faceted sphere), we can generate a standard icosphere mesh. Then, during vertex creation:

Java

```
float noiseValue = fastNoise.GetNoise(vertex.x, vertex.y, vertex.z);
float displacement = 1.0f + (noiseValue * chaosFactor);
```

`vertex.scl(displacement);`

This displaces the vertices based on 3D noise, creating a lumpy, irregular object. When this mesh is used to create a btGImpactMeshShape in Bullet, the physical rolling behavior becomes unpredictable and chaotic, perfectly matching the "Eldritch Math" theme of the early game.

4.4. Synergies with Existing Systems

The proposed overhaul interacts deeply with other planned features found in the design docs.

- Dynamic Weather [1]: The design doc mentions "Rain" and "Storms."
  - *Synergy:* Rain affects the physics simulation. If it is raining in the game world, the "Dice Tray" is wet. Friction is reduced by 50%. Thunderclaps [1] cause the tray to vibrate, potentially saving a bad roll or ruining a good one.
- Gore System [1]: The gore system uses BloodParticle and SurfaceDecal.[1]
  - *Synergy:* If a "Blood Price" wager is made, the dice literally bleed on the tray. The GoreManager [1] can be reused to render blood splatters *inside* the dice tray viewport, staining the felt and affecting the "readability" of the dice faces (obscuring numbers).
- Spawn Logic [1]: The spawn logic uses "Budgets."
  - *Synergy:* We can introduce "Math Shrines" as a spawnable entity. These shrines act as tutorial stations, allowing players to test their equations on dummy targets without spending resources.

---

5. Detailed Item Synergy Tables

To fulfill the user's request for specific, "bizarre" synergies, we define the following interaction tables. These should be implemented via a SynergyManager class that listens to inventory changes.

Table 1: The Elemental Constants

Item Name	Math Concept	Base Effect	Synergy with Gravity (π)	Synergy with Speed (c)
Potion of π	Transcendental Numbers	**Infinite HP / Infinite Mass.** Player cannot move.	**Singularity Event:** Nearby enemies are pulled into the player's tile and crushed (Instant Kill).	**Time Stop:** Because Mass is infinite, time dilation occurs. Enemies stop moving.
Bow of Light (c)	Speed of Light	**Instant Hit.** Projectile velocity is max float.	**Relativistic Mass:** Arrows gain mass as they travel. Max range shots deal massive knockback.	**Tachyonic Antitelephone:** Arrows hit the enemy *before* the turn starts (Pre-emptive strike).
Euler's Shield	$e^{i\pi} + 1 = 0$	**The Zero State.** 5% chance to negate damage.	**The Identity:** If HP, Mana, and XP end in $1, 0, 0$, invincibility is permanent.	**Energy Conversion:** Absorb incoming damage and convert it to Speed.

Table 2: Asymptotic Weaponry

Weapon Class	Complexity	Scaling Logic	Best Use Case
Dagger of $O(1)$	Constant Time	Always deals 10 Damage. Unaffected by	High-Armor / High-Evasion enemies (e.g.,

		armor or buffs.	Wraiths [1]).
Sword of $O(n)$	Linear Time	Damage = Turn Count $\times$ 2.	Standard combat encounters.
Hammer of $O(n^2)$	Quadratic Time	Damage = (Turn Count)^2.	Bosses. Weak start, but ramps up rapidly.
Staff of $O(2^n)$	Exponential	Damage doubles every hit. Resets on miss.	"Gambler" builds. Requires spending Food to ensure hits.
Wand of $O(n!)$	Factorial	$1, 2, 6, 24, 120...$	The ultimate risk. Surviving 6 turns guarantees a kill on almost anything.

---

6. Conclusion and Implementation Roadmap

The "Axiomatic Engine" overhaul transforms *Tarmin2Revised* from a nostalgic remake into a sophisticated ludic experiment. By grounding the gameplay in the manipulation of mathematical truth—represented physically by chaotic dice and conceptually by equation-based itemization—we create a unique value proposition for the "math nerd" demographic.

Implementation Roadmap:

1. **Phase 1: Physics Core:** Integrate gdx-bullet. Create the DiceTray scene and implementing the secondary ModelBatch render pass in GameScreen.
2. **Phase 2: The Math Data Layer:** Refactor ItemModifier to support Equation types. Update items.json [1] with the new mathematical artifacts.
3. **Phase 3: The Narrative UI:** Implement the EquationRenderer in Hud.java. Add the "Decoder Ring" mechanic to translate eldritch runes into readable math.
4. **Phase 4: Content Integration:** Procedurally generate the "Impossible Dice" meshes

using FastNoiseLite. Script the "Minotaur" fight to require balancing an equation rather than just depleting HP.

This approach leverages the full potential of the existing codebase—rendering, procedural generation, and stats—while pushing the boundaries of what a dungeon crawler can be. The dungeon is no longer just a maze of stone; it is a maze of logic, waiting to be solved.

Data Sources Referenced:

- **Codebase:** Tarmin2.java [1], CombatManager.java [1], GameScreen.java [1], FirstPersonRenderer.java [1], LootTable.java [1], ItemModifier.java [1], PlayerStats.java [1], FastNoiseLite.java [1], GoreManager.java [1], Hud.java.[1]
- **Design Documents:** Dynamic Weather [1], Fog of War [1], Gore System [1], Spawn Logic [1], Ranged Combat.[1]
- **Assets:** items.json [1], monsters.json [1], Shaders (crt.frag).[1]

Works cited

1. bluephoenixmedia/tarmin2revised