

Project Introduction / Summary

In this project, I am developing a top-down rogue-lite action game inspired by popular indie titles like *Brotato* and *Vampire Survivors*. These games are known for their fast-paced, wave-based combat, where players survive against hordes of enemies using automatically triggered attacks and randomized upgrades. The appeal lies in their simple control schemes, escalating difficulty, and the satisfying loop of quick decision-making and visual feedback. I aim to capture that same engaging experience with a game that is lightweight, visually appealing, and fun to play.

The game, which I am calling **Infinite Arena**, will challenge players to survive as long as possible in an endless digital battlefield. My primary goals are to create a polished and responsive gameplay loop, deliver a clear sense of progression through upgrades and enemy scaling, and build a visually rich environment that enhances player engagement. To achieve these goals, I plan to use the Godot engine to develop all core gameplay systems from scratch, including player control, enemy spawning, health/damage logic, and a wave progression system. I will also focus on integrating custom visual and audio effects that reinforce the game's energy and intensity. By the end of the project, I aim to have a fully playable and well-balanced prototype that showcases thoughtful game design and a compelling player experience.

Repository URL: <https://github.com/csdatnguyen/infinite-arena.git>

Primary Aesthetics

Infinite Arena will primarily focus on the aesthetics of **Challenge** and **Sensation** to create an experience that is both mentally engaging and visually stimulating.

The aesthetic of **Challenge** will be central to the player's experience, positioning the game as an ongoing test of endurance, reflexes, and adaptability. Players will face an ever-increasing number of enemies, each wave introducing new patterns, speeds, or behaviors. The tension builds as the space grows more chaotic, forcing quick decisions under pressure. This obstacle-course-like progression reinforces the core rogue-lite design, where skillful play and learning through repetition are key to survival. The satisfaction of overcoming difficult waves will serve as the primary source of player motivation and engagement.

The aesthetic of **Sensation** will enhance this challenge through visual and auditory feedback. Custom visual effects such as screen shakes on damage, explosive animations, and hit impact visuals will contribute to a more immersive and engaging gameplay experience. The game will also use audio elements like rhythmic background music and sharp sound cues to emphasize intensity and impact. These elements are designed to make each moment of combat feel more intense and reactive, reinforcing the stakes of the challenge and deepening the player's connection to the game environment.

By focusing on Challenge and Sensation, *Infinite Arena* aims to create an experience that is both strategically engaging and sensorially rich—keeping the player not just engaged, but fully absorbed in the action.

Primary Mechanics

One of the core mechanics in *Infinite Arena* is the **auto-firing combat system**. Instead of relying on manual button presses to attack, the player's weapon fires automatically at regular intervals. The direction of fire may vary depending on the player's setup—it could be aimed at the nearest enemy or directed toward the mouse position. This flexibility allows for different combat styles and player experiences. By removing the need for constant attack input, the mechanic encourages players to focus on positioning, movement, and evasion—key elements in surviving against large waves of enemies. It also contributes to the game's strategic depth, as players must navigate the arena efficiently while maximizing the impact of their attacks in real time.

Supporting this combat loop is the **wave-based enemy spawning system**. Enemies appear in increasingly difficult waves, with each new wave introducing greater numbers, faster speeds, or unique behaviors. This progressive system ensures that players are constantly challenged and must adapt quickly as pressure builds. It also creates a structured rhythm to gameplay, balancing short-term survival with long-term endurance and adding momentum to each run.

To enhance adaptability and replayability, *Infinite Arena* includes **randomized power-ups and upgrades**. These items are dropped by defeated enemies and can modify stats such as movement speed, fire rate, health, or even give special abilities. The randomness introduces an element of unpredictability that invites players to experiment with different combinations and playstyles. This mechanic also enhances the game's replay value, as each run can feel different depending on the power-ups acquired and how the player chooses to use them.

Another key mechanic is the **dash system**, which gives players a burst of mobility and a brief window of invulnerability. When activated, the player dashes in their current movement direction, allowing them to evade attacks or reposition quickly. This ability is limited by a cooldown timer, which prevents it from being overused and encourages strategic timing. The dash mechanic adds depth to the movement system, rewards quick reflexes, and enhances the game's overall pace and intensity.

Finally, the **health and collision system** introduces clear consequences and risk-reward dynamics. The player has a limited health pool and must avoid direct contact with enemies or their attacks. Each hit brings them closer to failure, creating tension and drives the need for careful movement and situational awareness. Combined with visual feedback such as screen shake and hit effects, this system keeps the stakes high and ensures that every decision feels meaningful in the heat of battle.

Alpha Reflection

Since the initial checkpoint, the project has progressed significantly from concept to implementation. At the proposal stage, Infinite Arena existed only as a design vision, with core mechanics and aesthetics outlined but no functioning systems in place. As of the Alpha milestone, key gameplay features are now operational, including the auto-firing weapon system, enemy spawning system, health system, and collision detection. Core systems such as player movement, enemy AI tracking, damage handling, and game-over logic are integrated and working together to form a complete gameplay loop. This marks a major step forward from the planning phase, turning theoretical mechanics into practical, testable features.

Along the way, several additions and refinements were made to improve design clarity and code organization. Notably, creational patterns like Factory Method and Lazy Initialization were adopted to manage object instantiation more cleanly, while behavioral patterns like Observer allowed for decoupled game flow management through signals. I also deepened my understanding of Godot's physics and animation systems, using them to enhance both player feedback and enemy behavior. One key lesson learned was the importance of keeping gameplay systems modular and reusable early on — this made it easier to debug, iterate, and prepare for future expansion. Overall, the Alpha phase has laid a solid foundation for polishing, balancing, and adding content in the next stages of development.

Software Elements

1. Creational Pattern - Factory Method

To streamline object creation and improve code modularity, I implemented the Factory Method pattern in two key areas: bullet creation and mob spawning. Instead of directly instantiating scenes within gameplay scripts, I created dedicated factory scripts—**bullet_factory.gd** and **mob_factory.gd**—which encapsulate the instantiation logic. This approach clarifies where and how new game objects are created, making future modifications more manageable. For example, in **gun.gd**, the *shoot()* function now uses *bullet_factory.create_bullet()* to generate new bullets. Similarly, in *survivors_game.gd*, the *spawn_mob()* function uses *mob_factory.create_mob()* to spawn enemies. This pattern supports scalability and reinforces a clean separation of concerns between gameplay logic and object creation.

2. Creational Pattern – Lazy Initialization

In **mob.gd**, I applied the Lazy Initialization pattern to optimize how the smoke explosion effect is loaded when a mob is destroyed. Rather than preloading the smoke scene at the top of the script or instantiating it repeatedly in the *take_damage()* function, I created a helper function called *get_smoke_scene()* within the same file. This function checks whether the resource has already been loaded; if not, it loads it and caches it for future use. The function is called from within *take_damage()* only when the mob's health reaches zero. This approach ensures the smoke effect is only loaded the first time it's needed, improving memory efficiency and runtime performance as more mobs are spawned and destroyed during gameplay.

3. Behavioral Pattern – Observer

The Observer pattern is used to manage game-over conditions in a decoupled and flexible way. In **player.gd**, I declared a custom *signal health_depleted*, which is emitted when the player's health reaches zero. This signal is connected in **survivors_game.gd** within the *_ready()* function. Once connected, the *_on_player_health_depleted()* function is triggered, which activates the Game Over screen and pauses the game. By using signals, the player object does not need to know about the rest of the game's state or flow — it simply broadcasts an event, and other systems respond as needed. This clean separation improves modularity and makes future expansion easier to implement.

4. Game Engine Tool – Physics Simulation

Godot's physics engine is used extensively to handle core movement and collision systems in the game. In **player.gd** and **mob.gd**, *move_and_slide()* enables smooth, physics-based movement, with adjustable speed values that directly affect gameplay responsiveness. The player also uses *get_overlapping_bodies()* to detect enemy contact and apply damage in real time. In **bullet.gd**, bullets move using a *SPEED* constant multiplied by *delta* to maintain consistent frame-rate-independent motion, and rely on *Area2D* collision detection via *_on_body_entered()* to trigger impact effects. These physics-based systems make movement and combat feel responsive and grounded, reinforcing the game's focus on positioning and reflexes.

5. Game Engine Tool – Animation System

Animations are used throughout the project to enhance clarity, responsiveness, and visual feedback during gameplay. In **player.gd**, the character switches between walking and idle animations based on input velocity, by calling *play_walk_animation()* and *play_idle_animation()* from **happy_boo.gd**. Similarly, in **mob.gd**, enemies trigger *play_walk()* and *play_hurt()* functions defined in **slime.gd**, depending on their movement and state. These animations are controlled using Godot's AnimationPlayer node and are triggered programmatically in response to real-time game events. This dynamic use of animation reinforces both the Challenge and Sensation aesthetics by providing clear, satisfying feedback to player and enemy actions.