Dat Nguyen
CPSC 5270 01 25SQ Graphics/Game Project
Beta Report

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 Aren*a 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.

Beta Reflection

Since the Alpha checkpoint, *Infinite Arena* has made substantial progress in gameplay structure, visual feedback, and system flexibility. Several new design patterns have been introduced, such as the Builder pattern to support more modular wave creation and the State pattern to better manage enemy behavior. These additions have improved both code maintainability and gameplay clarity. The Prototype pattern allowed for easy creation of enemy variants, further enhancing variety and replayability. Overall, the project has transitioned from focusing purely on core mechanics to emphasizing polish, balance, and scalability.

I also began incorporating visual enhancements such as shaders to improve combat feedback and distinguish enemy types more clearly. These features were not part of the original plan but became necessary as gameplay complexity increased. Through this phase, I've learned the importance of designing systems that are flexible and reusable, especially as the game expands. The shift toward a more modular and pattern-driven approach has made it easier to iterate on features and prepare for future polish. This stage marks a turning point where the game feels more cohesive and closer to the intended final experience.

Software Elements
[1 to 5 added during Alpha]
[6 to 10 added during Beta]

1. Creational Pattern - Factory Method
To streamline enemy creation and maintain modular code structure, I implemented the Factory Method pattern in **mob_factory.gd** using multiple specialized methods. Rather than directly instantiating mobs within gameplay scripts, the factory provides functions such as *create_base_mob()*, *create_small_mob()*, and *create_big_mob()*. Each method instantiates a shared base mob scene and then customizes its properties—including *scale*, *speed*, *health*, and *shader tint*—to produce the desired variant. This encapsulates the configuration logic within the factory itself and keeps enemy creation consistent and centralized. The factory is used by **wave_builder.gd**, which dynamically selects the appropriate mob type based on the current wave configuration, supporting scalable and flexible wave generation.

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.

6. Behavioral Pattern - State
To better manage enemy behavior and improve code clarity, I implemented the State Pattern in **mob.gd** using an *enum* called *MobState* with three defined states: *IDLE*, *CHASE*, and *DEAD*. The enemy's behavior in *_physics_process(delta)* is controlled through a *match* statement, with each state determining how the mob moves and reacts. In the *IDLE* state, the mob remains stationary and plays an idle animation. Once the player is within a specified range, the state transitions to *CHASE*, prompting the mob to follow the player using directional movement and the Godot physics engine. When the mob's health reaches zero, the state changes to *DEAD*, triggering a one-time smoke explosion using lazy initialization and then removing the mob from the scene. This approach simplifies state transitions, reduces conditional clutter, and allows for easier expansion of behaviors in future updates.

7. Creational Pattern - Prototype

The Prototype Pattern is applied to efficiently create multiple mob variants from a shared template. In **mob_factory.gd**, the base mob scene (**mob.tscn**) is preloaded and instantiated as the starting point for all mob types. This base instance is then customized to create distinct behaviors by modifying properties such as *scale*, *speed*, *health*, and *shader tint*. For example, *create_small_mob()* produces a fast, low-health mob scaled down in size, while *create_big_mob()* generates a larger, slower, and more durable variant. Visual differentiation is handled by the *set_tint()* function, which adjusts shader parameters on the mob's **Slime/Anchor/SlimeBody** node. By cloning and modifying a common prototype, this approach avoids redundancy and keeps mob configuration consistent, scalable, and easy to maintain.

8. Creational Pattern - Builder
The enemy wave system was restructured using the Builder Pattern to allow for more readable and configurable wave generation. In **wave_builder.gd**, the builder exposes setter methods like *set_mob_count()*, *set_mob_type()*, and *set_spawn_area()*, which can be chained together before calling *build_wave()* to construct and spawn the configured wave. The *create_mob()* function handles which type of mob to create—*base*, *small*, or *big*—by calling the appropriate method from **mob_factory.gd**. After creation, each mob is passed to *spawn_mob_randomly()*, which places the mob at a random location along a path using *PathFollow2D.* This design is used in **survivors_game.gd** to dynamically generate different types of waves based on a predefined list of configurations.

9. Shader - Color Tint for Mob Variants
To help visually distinguish different types of enemies, I customized a shader to apply color tinting based on mob type. The shader, written in **slime.gdshader**, includes two uniforms—*base_tint_color* and *tint_strength*—which allow each mob instance to be tinted independently. For example, small mobs are tinted cyan, large mobs are tinted pink, and base mobs remain untinted. These tint values are applied dynamically in **mob_factory.gd** using the *set_tint()* function after a mob is instantiated. Additionally, in **mob.gd**, the shader material is duplicated using *duplicate()* and reassigned in *_ready()* to prevent shared tinting effects and ensures each mob has an isolated material instance. This approach enhances visual clarity during combat and supports the game's Sensation aesthetic by making enemy types immediately recognizable through color cues.

10. Shader - Flash on Hit Effect
Instant visual feedback during combat is achieved through a flash-on-hit effect controlled by a custom shader in **slime.gdshader**. The shader uses a *flash_amount* uniform to blend the mob's sprite with a red *flash_color*, creating a quick flash when damage is taken. In **mob.gd**, the *flash()* function triggers this effect by setting *flash_amount* to *1.0*, waiting briefly with a timer, and then resetting it back to *0.0*. This function is called within *take_damage()* whenever the mob is hit. Since each mob has a duplicated shader material assigned in *_ready()*, the flash effect remains isolated to the affected instance. This sharp visual reaction helps reinforce the sensation of impact and improves player awareness during fast-paced encounters.