Dat Nguyen
CPSC 5270 01 25SQ Graphics/Game Project
Final 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 explosive animations and hit impact visuals will contribute to a more immersive and engaging gameplay experience. The game will also use audio elements like 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 to escape tough situations. 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, 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.

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 (**mob tint** and **flash on hit effect**) 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.

Final Reflection

Since the Beta checkpoint, *Infinite Arena* has evolved with a stronger emphasis on polish, responsiveness, and modularity. Several mechanics were finally implemented, including a player **dash system** that adds mobility and tactical depth, and a **Power-Up Manager** that enables stat-based upgrades using the **Facade** pattern. In addition, the **Strategy** pattern was applied to **enemy movement**, allowing different mob types to follow distinct behaviors such as zig-zag or straight-line chasing. The **Fluent Interface** pattern was also introduced to improve the readability and flexibility of **wave configuration**; by enabling method chaining, wave parameters can be set in a clean, expressive way.

On the visual side, shader enhancements were added to improve clarity and feedback—**bullets** now feature a bold **black outline** for better visibility, and the **player flashes red** with a **pulsating glow** effect when taking damage to emphasize urgency. A **sine wave UV distortion** shader was also applied to trees to simulate gentle **wind sway**, adding ambient motion to the environment.

In addition, a **knockback** effect was added to enemies when hit by bullets, reinforcing the game's use of **physics** for responsive combat feedback. **Audio** was also integrated with a satisfying laser **sound effect** each time the player fires, which enhances responsiveness and immersion.

One thing to note is that due to time constraints, power-ups were not implemented through dropped items as originally planned. However, to simulate the upgrade process and demonstrate the system's structure, upgrades were made accessible via key inputs—pressing "U" increases attack speed and "I" enlarges bullet size and hitbox. These upgrades are routed through a centralized Power-Up Manager using the Facade pattern, allowing clean separation from gameplay logic and making the system easy to expand in the future.

Also, initially, I intended to include a settings panel that would allow players to toggle between different shooting styles through the UI. However, due to time limitations, I implemented a more lightweight approach by enabling players to switch shooting modes using the "O" key. The player can alternate between two modes: targeting the closest enemy or aiming in the direction of the mouse cursor. While the feature isn't as fully developed as originally envisioned, it still successfully demonstrates the core idea of offering two distinct playstyles for players to choose from.

Another key revision was made to the **Factory Method** pattern used for enemy creation, which is now integrated with the **Prototype** pattern to improve efficiency and consistency. Initially, mobs were spawned through a generic *create_mob()* function. This was later refactored into a more structured and optimized system in **mob_factory.gd**, where methods like *create_base_mob()*, *create_small_mob()*, and *create_big_mob()* each return a customized clone of a cached prototype mob using *.duplicate()*. This approach allows all variants to share a common base while being tailored with different *scale*, *speed*, *health*, *movement strategy*, and *shader tint*. By combining **Factory** and **Prototype**, the code cleanly separates creation logic from usage, avoids redundancy, and ensures consistent configuration across mob types. These factory methods are now used in **wave_builder.gd** to construct scalable and varied waves, making the system modular, performant, and easier to maintain.

In addition to refining creation logic, the **State Pattern** implementation for enemy behavior was completely restructured for better modularity and scalability. Instead of relying on an *enum* and a *match* statement within **mob.gd**, state-specific logic is now encapsulated in separate script files—**idle_state.gd**, **chase_state.gd**, **and dead_state.gd**—all of which extend a shared base class **MobState**. Each state handles its own *enter()* and *update()* behavior, allowing clean separation of responsibilities and making transitions more intuitive through a centralized *change_state()* function in **mob.gd**. This refactor eliminates conditional clutter, simplifies the addition of new behaviors, and better aligns with the **State Pattern**'s principles by treating each state as a self-contained object.

As a final step, I took time to revisit and refactor my codebase to ensure better organization and maintainability. This included grouping related files into clearly named subfolders—for example, placing all state-related scripts into a **MobStates** folder, all movement strategies into a **MobMovement** folder, and all mob creation scripts such as **mob_factory.gd** and **wave_builder.gd** into a **MobCreation** folder within the main **mob** directory. In addition to restructuring the file system, I also added consistent and detailed comments across all scripts to clarify logic, document design pattern usage, and improve code readability. These finishing touches helped clean up the project structure, making it easier to understand, debug, and expand in future iterations.

One of the biggest lessons I learned was the value of taking one step at a time—it made the development process more manageable and significantly easier to debug. Although applying design patterns required more initial effort, it ultimately led to a more scalable and maintainable codebase. I also realized the importance of documenting code and organizing files properly; having clean, well-structured scripts made it much easier to navigate through the project and implement improvements later on.

Software Elements

Creational Programming Patterns

1. Factory Method + Prototype – Enemy Creation

To support modular and scalable enemy creation, I combined the Factory Method and Prototype design patterns in **mob_factory.gd**. A base mob scene (**mob.tscn**) is pre-instantiated once and stored as a prototype (*prototype_mob*). This prototype is then duplicated using *duplicate()* whenever a new enemy is created, avoiding repeated instantiation and improving performance. The factory exposes specialized methods—*create_base_mob()*, *create_small_mob()*, and *create_big_mob()*—each of which clones the prototype and customizes it with unique properties such as *scale*, *health*, *speed*, *movement strategy*, and *shader tint*. This hybrid approach encapsulates enemy configuration logic while efficiently reusing a shared template, keeping enemy creation consistent, memory-efficient, and easy to maintain. The factory methods are 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. Builder – Mob Wave Spawning
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.

3. Lazy Initialization – Smoke Loading
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 preloads it and caches it for future use. The smoke effect is instantiated in **dead_state.gd**, specifically within the *enter()* function, which is triggered when the mob transitions into the *DeadState*. 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.

Structural Programming Patterns

1. Facade – Centralized Power-Up System
To simplify how power-ups are applied to the player and to centralize upgrade logic, I implemented the Facade pattern through a script called **powerUpManager.gd**. This script provides a single entry point via the *apply_upgrade(player, upgrade_type)* function, which handles different upgrade effects behind the scenes. For example, when the player presses the *"U"* key, it triggers the *"IncreaseFireRate"* upgrade, which accesses the player's gun and reduces its internal *fire_rate* variable by 10% through the *set_fire_rate()* function in **gun.gd**. Similarly, pressing *"I"* triggers the *"IncreaseBulletSize"* upgrade, which calls *set_bullet_scale()* in the same script to enlarge future bullets up to a capped size. The power-up logic is kept out of the main gameplay loop and instead routed through this facade, keeping the game code clean, maintainable, and scalable for future upgrades. This approach made it easy to simulate upgrade pickups and integrate enhancements without tightly coupling gameplay systems.

Behavioral Programming Patterns

1. Observer – Game Over Trigger via Signals

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.

2. State – Mob Behavior Management

To better manage enemy behavior in a modular and extensible way, I implemented the State Pattern in **mob.gd** by introducing separate state classes—**idle_state.gd**, **chase_state.gd**, and **dead_state.gd**—all of which extend from a shared **mob_state.gd** base class. Each mob holds a *current_state* variable of type *MobState*, which defines two key methods: *enter(mob)* and *update(mob, delta)*. In *_ready()*, the mob starts in the *IdleState*, and in each *_physics_process(delta)* tick, it delegates logic to the current state's *update()* method. For example, *IdleState* monitors the player's proximity and transitions to *ChaseState* when close enough, while *ChaseState* switches to *DeadState* when the mob's *health* drops to zero. Transitions are handled by the *change_state()* method, which assigns the new state and calls its *enter()* logic. This structure encapsulates behaviors cleanly in their respective files, eliminates condition-heavy logic, and makes it easier to introduce new states or tweak behavior independently.

3. Strategy – Modular Enemy Movement

To support flexible and varied enemy behaviors, I implemented the Strategy pattern to control mob movement using interchangeable strategy scripts. Instead of hardcoding logic in the mob itself, each mob is assigned a movement strategy via the *movement_strategy* variable in **mob.gd**, which is invoked during the *CHASE* state through the *move(mob, delta)* function. These strategies inherit from the base class **mobMovementStrategy.gd** and are defined in separate files such as **straightMovementStrategy.gd** and **ZigZagMovement.gd**. The zig-zag movement strategy uses a sine wave to create lateral offsets, making the enemy's path more unpredictable. It includes adjustable parameters like *zigzag_amplitude*, *zigzag_frequency*, and *time*, which control how wide and how fast the side-to-side movement is. The straight movement strategy, on the other hand, simply calculates the normalized direction from the mob to the player and applies velocity accordingly. Strategy assignment is handled in **mob_factory.gd**, where small mobs receive the zig-zag pattern while base and big mobs use straight movement. This separation of logic makes enemy behavior more modular, customizable, and easy to expand with new movement styles in the future.

4. Fluent Interface - Chainable Wave Configuration

The Fluent Interface pattern is used in **wave_builder.gd** to enable readable and expressive method chaining when configuring and spawning enemy waves. This design allows multiple setter functions—such as *set_mob_count(count: int)*, *set_mob_type(type: String)*, and *set_spawn_area(area: Node)*—to each return self, enabling them to be chained together in a single expression. In **survivors_game.gd**, this fluent interface is demonstrated when spawning a wave using the following code block:

```
(
	wave_builder
	.set_mob_count(config["count"])
	.set_mob_type(config["type"])
	.set_spawn_area(self)
	.build_wave()
)
```

This chaining style simplifies wave construction and improves code readability by expressing configuration and execution in a clear sequence. It works alongside the Builder Pattern, reinforcing modular wave setup with a fluent, behavior-driven syntax.

Shaders

1. 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.

2. Flash on Hit Effect

A flash-on-hit effect is applied to the player character to provide immediate and reactive visual feedback when taking damage. Implemented in **player_shader.gdshader**, the shader uses uniforms *flash_amount*, *flash_color*, and *player_color* to control the visual output. In **player.gd**, when the player overlaps with mobs, the *is_taking_damage* flag is triggered and the *flash()* function updates the shader's *flash_amount* parameter. When active, the shader blends a pulsing red glow over the player sprite by modulating *flash_color* with *sin(TIME * 20.0)*, creating a rhythmic brightness effect (*flash_glow*). This effect not only turns the player red but also adds a pulsating glow that visually amplifies the sense of danger and urgency. When not flashing, the sprite reverts to its normal tinted appearance using *player_color*. This shader-driven reaction makes the player visually stand out during moments of danger and enhances sensory feedback, aligning with the game's Sensation aesthetic.

3. Wind Sway via Sine Wave UV Distortion
To enhance environmental immersion, I implemented a sine wave UV distortion shader applied to the pine tree material in **tree_shader.gdshader**. The shader creates a subtle wind sway effect by offsetting the texture's UV coordinates horizontally using the sine of the vertical UV position and the built-in *TIME* variable. Specifically, the *UV.x* value is adjusted by *sin(UV.y * frequency + TIME * speed) * intensity*, where *frequency*, *speed*, and *intensity* are uniform parameters that control the sway behavior. This effect creates smooth, rhythmic movement that mimics natural wind without the need for additional animations or textures. It adds atmosphere to the environment while remaining lightweight and easily adjustable.

4. Alpha-Based Outline Shader
To improve visual clarity during combat, especially for small and fast-moving projectiles, I implemented an alpha-based outline shader for bullets. The shader, located in **bullet_shader.gdshader**, enhances bullet visibility by outlining the bullet sprite's edges using a uniform *outline_color* (default black) and adjustable *outline_width*. It works by checking transparent pixels and their neighbors; if a transparent pixel borders an opaque one, it gets colored with the outline. This technique ensures that bullets, which are initially small and move quickly, remain easy to see against any background and clearly indicate their hitbox. The shader is applied to the bullet's *Projectile* node inside **bullet.tscn**, helping maintain visual clarity and reinforcing the game's Sensation aesthetic.

Game Engine Tools/ Technologies

1. Physics System

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.

To enhance the physical responsiveness of combat, a knockback effect was implemented for enemies upon bullet collision. In **bullet.gd**, the *_on_body_entered()* function checks whether the impacted enemy has an *apply_knockback()* method. If so, it calculates a direction vector from the bullet to the enemy and applies a force to simulate the impact. This force is stored as *knockback_velocity* in **mob.gd** and applied to the mob's position each frame, gradually easing off using *move_toward()* to simulate friction. While not using rigid body dynamics, this implementation captures the feel of physics-based motion and gives each hit a satisfying sense of impact, making combat interactions more dynamic and believable.

2. Audio Output

Audio was incorporated into the game to enhance player feedback and overall immersion during gameplay. A shooting sound effect was added using Godot's built-in *AudioStreamPlayer2D* node, which is triggered every time the gun fires. The sound file, **laser_bullet.wav**, was assigned to a dedicated audio node under the Gun scene. Playback was initiated programmatically in the **gun.gd** script within the *shoot()* function using *$ShootSound.play()*. This integration gives each shot a satisfying auditory cue, reinforcing the player's actions and improving the game's responsiveness. Adding this simple yet effective audio feature showcases how sound design can elevate gameplay without requiring complex implementation.