Space Shooter Game - Technical Documentation

Table of Contents

Game Overview

This is a 2D top-down space shooter built in Godot 4.3, featuring wave-based gameplay where players pilot customizable ships against the Red Fleet enemy forces. The game includes RPG-like progression through equipment upgrades, multiple ship types, and increasingly challenging waves.

Core Gameplay Loop

1. **Combat Phase**: Fight waves of enemies while collecting scrap
2. **Shop Phase**: Purchase upgrades between waves using collected money
3. **Progression**: Unlock stronger ships and equipment to handle harder waves

Core Architecture

Global Singletons

The game uses several autoload scripts that serve as global data managers:

`Player.gd` - Central Player Data Hub

```gdscript
```

```
# Core player statistics and attributes
var Player_Attributes = {
    player_hp_total = 100,
    player_current_HP = 100,
    player_weapon_dmg = 1,
    # ... extensive attribute system
}

var Player_Stats = {
    health = 0,
    armor = 0,
    damage = 0,
    # ... upgrade statistics
}
```

Purpose: Stores all player data including health, weapons, stats, and game progress. Acts as the single source of truth for player state.

`RedFleet.gd` - Enemy Fleet Management

gdscript

```
var Fleet_Attributes = {
    tier_1 = {dmg = 5, hp = 5, spd = 75-175},
    tier_2 = {dmg = 25, hp = 15, spd = 50-125},
    # ... enemy tier definitions
}
```

Purpose: Defines enemy statistics and manages fleet-wide state like enemy counts and end-of-wave conditions.

`Powerups.gd` - Ship & Weapon Systems

Purpose: Manages ship selection and applies ship-specific bonuses to player stats.

`Shop.gd` - Item Definitions

Purpose: Contains all shop items, their effects, and pricing information.

Player Systems

Player Controller (`character_body_2d.gd`)

The main player controller handles:

- **Movement**: 8-directional movement with speed modifiers
- **Shooting**: Weapon firing with cooldown management
- **Animation**: Dynamic sprite changes based on movement and damage states
- **Mini-bot Management**: Spawns and controls orbiting AI companions

Key Features:

- **Multi-bolt System**: Chaos stat allows firing multiple projectiles
- **Regeneration**: Automatic HP recovery over time
- **Life Steal**: Healing from dealing damage to enemies
- **Invincibility Frames**: Brief immunity after taking damage

Health & Shield System (`healthbar.gd`)

```gdscript
# Efficient update system - only updates when values change
func update_health_if_changed():
    var current_health = Player.Player_Attributes.player_current_HP
    if current_health != previous_health:
        health = current_health
        previous_health = current_health
```

Features animated damage bars and efficient UI updates that only occur when values actually change.

Mini-bot Companions (`mini_bot.gd`)

- **Orbital Movement**: Bots orbit around the player
- **Synchronized Shooting**: Fire when player shoots
- **Scaling System**: Number of bots based on AI Integration stat

Magnet System (`magnet.gd`, `area_2d.gd`)

- **Visual Feedback**: Pulsing effect when active
- **Size Scaling**: Area increases with mag_size stat
- **Automatic Collection**: Pulls scrap toward player

Enemy Systems

Enemy Types

Fighter (`fghtr_1_bod.gd`)

Basic enemy with enhanced movement patterns:

- **Simple Side-to-Side**: Basic collision-based movement
- **Sine Wave**: Smooth wave motion across screen
- **Aggressive Weave**: Sharp direction changes
- **Spiral Descent**: Spiral motion while falling
- **Erratic Dart**: Quick random direction bursts

Movement complexity increases with wave number.

Bomber (`bomber_1_bod.gd`)

Slower, more dangerous enemy:

- **Directional Movement**: Left/right movement with collision detection
- **Heavy Projectiles**: Fires slower but more damaging bullets
- **Higher Rewards**: Drops tier-2 scrap

Cigar Bot (`cigar_bot.gd`)

Advanced enemy with unique behavior:

- **Vertical Oscillation**: Moves up and down in patterns
- **Multi-position Shooting**: Fires from multiple weapon points
- **Burst Fire**: Shoots 3 bullets in quick succession
- **Direction Detection**: Uses raycasting for wall detection

Mine (`mine.gd`)

Passive threat that activates on proximity:

- **Falling State**: Normal downward movement
- **Chasing State**: Activated when player enters detection area
- **Contact Damage**: Explodes on player contact
- **Visual Effects**: Rotation and pulsing red effect

Enemy Spawning System

Main Spawner (`spawner.gd`)

```gdscript
# Wave-based spawning limits
const BASE_FIGHTER_LIMIT = 8
const FIGHTER_LIMIT_INCREASE = 2

func update_fighter_limit():
    if current_wave == 1:
        fighter_limit = BASE_FIGHTER_LIMIT
    else:
        fighter_limit = BASE_FIGHTER_LIMIT + (current_wave - 1.5) * FIGHTER_LIMIT_INCREASE
```

Features:

- **Dynamic Limits**: Enemy spawn limits increase with wave progression
- **Mixed Spawning**: 80% fighters, 20% mines
- **Carrier Management**: Special spawning logic for high-tier enemies

Specialized Spawners

- **Bomber Spawner** (`spawner_Bombers.gd`): Wave 2+ bomber spawning
- **Cigar Spawner** (`cigar_spawn_1.gd`): Wave 4+ special enemy spawning
- **Carrier Spawner** (`carrier_spawn.gd`): Wave 10+ boss-level spawning

Combat & Weapons

Player Weapons

Basic Bolt (`basic_bolt.gd`)

```gdscript
```

```
func _on_body_entered(body: CharacterBody2D):
    if body.has_method("take_damage"):
        damage = dmg
        body.take_damage(damage)

        # Life steal calculation
        if Player.Player_Attributes.leech > 0:
            var heal_amount = 0.5 + (Player.Player_Attributes.leech * 0.1)
            Player.Player_Attributes.player_current_HP = min(
                Player.Player_Attributes.player_current_HP + heal_amount,
                Player.Player_Attributes.player_hp_total
            )
```

Features:

- **Damage Scaling**: Damage increases with player stats
- **Life Steal**: Heals player based on damage dealt
- **Hit Registration**: Tracks accuracy statistics

Enemy Weapons

- **Basic Enemy Bolt**: Standard red projectiles
- **Bomber Bolt**: Slower, higher damage projectiles
- **Cigar Bullet**: Fast horizontal projectiles with vertical drop

Damage System

All entities use a standardized `take_damage(damage: float)` method:

1. **Reduce HP** by damage amount
2. **Update statistics** (damage dealt/taken)
3. **Visual feedback** (damage flash effect)
4. **Death handling** if HP reaches zero

Progression & Economy

Shop System (`shop.gd`)

Item Categories

- **Common**: Basic stat improvements (Health +1, Damage +1)

- **Uncommon**: Moderate improvements with trade-offs
- **Rare**: Significant bonuses with notable penalties
- **Exceptional**: Game-changing abilities (Magnet, Auto Blaster)

Pricing System

```gdscript
match item.rarity:
    "common":    prices[i] = 50 * Player.Wave / 2
    "uncommon":  prices[i] = 100 * Player.Wave / 2
    "rare":      prices[i] = 150 * Player.Wave / 2
    "exceptional": prices[i] = 200 * Player.Wave / 2
```

Prices scale with wave progression to maintain challenge.

Reroll System

- **Escalating Cost**: Each reroll costs more than the last
- **Wave Scaling**: Base cost increases with wave number

Stat Application (`shop.gd` - `player_char_update()`)

Complex system that converts shop upgrades into actual player attributes:

```gdscript
# Health scaling example
if Player.Player_Stats.health > 0:
    var upgrade = (Player.Player_Stats.health * 10) + Player.Player_Attributes.player_hp_total
    Player.Player_Attributes.player_hp_total = upgrade
```

Ship Selection System (`powerups.gd`)

Three distinct ship types with different playstyles:

- **Old Faithful**: Balanced, no bonuses or penalties
- **Royal Spear**: Fast interceptor (+Speed, +Shield, -Armor)
- **Drone Bus**: Support ship (+AI Integrations, -Speed, -Fire Rate)

UI & Menus

Menu Flow

```
Main Menu → Ship Selection → Level → Pause Menu
                                ↓          ↑
                          Level ← Shop ← Wave Complete
```

Dynamic UI Updates

Most UI elements use `_process()` to update in real-time:

```gdscript
func _process(delta: float) -> void:
    self.text = str('Wave:', Player.Wave)
```

Key UI Components

- **Health Bar**: Animated damage with shield overlay
- **Wave Timer**: Countdown to next wave
- **Stats Display**: Real-time player statistics
- **Game Over Screen**: Final statistics and restart options

Level Management

Wave System (`wave_timer.gd`)

```gdscript
func _process(delta: float) -> void:
    if Player.Wave == 1: wait_time = 20.0
    if Player.Wave == 2: wait_time = 30.0
    # ... escalating wave durations
```

Wave Progression:

- Waves 1-3: 20-35 seconds (learning phase)
- Waves 4-6: 40-50 seconds (difficulty ramp)
- Waves 7+: 60 seconds (sustained challenge)

Level Controller (`level_1.gd`)

Central coordinator that manages:

- **Scene transitions** between combat and shop

- **Spawner activation** and coordination
- **Background scrolling** and parallax effects
- **Game over conditions** and restart logic

Background Systems

- **Planet Spawner** (`planet_spawn_1.gd`): Procedural background objects
- **Parallax System** (`paralax_spawn.gd`): Layered scrolling backgrounds
- **Moving Planets** (`planets_body.gd`): Animated celestial bodies

Performance & Debug

Debug System (`debug.gd`)

Comprehensive performance monitoring:

```gdscript
# Real-time performance tracking
func get_performance_stats():
    var stats = {}
    stats["avg_fps"] = 1.0 / avg_frame_time
    stats["memory_peak"] = OS.get_static_memory_peak_usage()
    stats["render_info"] = {
        "visible_objects": Performance.get_monitor(Performance.RENDER_TOTAL_OBJECTS_IN_FRAME),
        # ... additional render statistics
    }
```

Features:

- **FPS Monitoring**: Real-time frame rate tracking
- **Memory Usage**: Peak memory consumption
- **Function Profiling**: Time specific code blocks
- **Node Tree Analysis**: Identify performance bottlenecks

Optimization Patterns

Timer Replacement Pattern

Many scripts use lifetime timers instead of Timer nodes:

```gdscript
```

```gdscript
# Instead of Timer nodes, use delta accumulation
var lifetime_timer: float = 0.0

func _physics_process(delta):
    lifetime_timer += delta
    if lifetime_timer >= max_lifetime:
        queue_free()
```

Efficient UI Updates

```gdscript
# Only update when values actually change
func update_health_if_changed():
    var current_health = Player.Player_Attributes.player_current_HP
    if current_health != previous_health:
        health = current_health
        previous_health = current_health
```

Code Organization

Script Categories

Global Systems (`*_*.gd`)

- **player.gd**: Player data and statistics
- **red_fleet.gd**: Enemy definitions and fleet state
- **powerups.gd**: Ship and weapon configurations
- **shop.gd**: Item definitions and shop logic

Entity Controllers (`*_bod.gd`, `character_body_2d.gd`)

- **character_body_2d.gd**: Player controller
- **fghtr_1_bod.gd**: Fighter enemy controller
- **bomber_1_bod.gd**: Bomber enemy controller

Weapon Systems (`*_bolt*.gd`)

- **basic_bolt.gd**: Player projectiles
- **basic_bolt_r_0.gd**: Enemy projectiles
- **mini_bot_bolt.gd**: Companion projectiles

UI Components (`*_label.gd`, `*_menu.gd`)

- Real-time display updates
- Menu navigation logic
- Game state management

Naming Conventions

- **Global Scripts**: Descriptive names (player.gd, shop.gd)
- **Entity Scripts**: Type + "bod" suffix for bodies
- **UI Scripts**: Element + function (wave_number.gd)
- **Weapon Scripts**: Type + "bolt" + variant

Data Flow Patterns

Centralized State Management

```gdscript
# Global state accessed throughout codebase
Player.Player_Attributes.player_current_HP
RedFleet.Fleet_Attributes.tier_1.hp
```

Event-Driven Updates

```gdscript
# UI responds to data changes
func _process(delta: float) -> void:
    self.text = str('Cash:', Player.money)
```

Signal-Based Communication

```gdscript
# Timers coordinate game phases
func _on_wave_timer_timeout() -> void:
    end_of_wave()
```

Development Notes

Key Design Decisions

1. **Global Singletons**: Simplified data access across scenes
2. **Timer Replacement**: Better performance than Timer nodes
3. **Modular Enemies**: Each enemy type in separate script
4. **Real-time UI**: Immediate feedback for player actions

Potential Improvements

1. **Signal System**: Reduce direct global access
2. **Component System**: Break down large entity scripts
3. **Resource Management**: Preload commonly used scenes
4. **Save System**: Persistent player progression

Performance Considerations

- **Object Pooling**: Consider for frequently spawned objects
- **LOD System**: Distance-based enemy simplification
- **Culling**: Remove off-screen entities
- **Batching**: Group similar draw calls

This documentation provides a comprehensive overview of the game's architecture and implementation. Each system is designed to work independently while contributing to the overall gameplay experience.