ChatGPT

Spectra: Unified Game Design Blueprint

Core Concept & Vision: Spectra is a physics-driven puzzle game where you "control a photon emission core navigating the instrument's optical modules" to restore a self-repairing spectrophotometer [1]. The player's goal is to **deliver photons** of specific wavelengths, energies, and polarizations to optical components to trigger reactions, reassemble modules, and gradually **restore full-spectrum functionality** [2]. Gameplay mixes precision laser puzzles with resource management and procedural upgrades (inspired by Portal, Opus Magnum, etc.) [3]. Puzzles emerge from real optical behavior: beams reflect, refract, split, and attenuate according to physical laws, and materials absorb or fluoresce based on their spectra [4] [5].

Gameplay Systems Overview: The game architecture is built around several interacting subsystems:

- **Photon Control:** The core player mechanics (move, aim, fire photons) define the **game feel** [6]. PhotonControl handles inputs (WASD movement, mouse/keys to aim & fire) and creates photon entities with given wavelength, intensity, and polarization [6] [7].
- **Optical Environment:** Levels (optical chambers) are populated with tiles/objects like mirrors, prisms, filters, polarizers, samples, absorbers, and detectors. These components **generate puzzles** by altering photon paths and states [8].
- **Spectroscopy Core:** Underlying all interactions is a physics engine translating real optics into game events. It computes beam energy ($E = 1240/\lambda$ in eV), handles attenuation (Beer–Lambert law), polarization effects, and interference [9] [10]. This system tracks beams' wavelength (λ), energy (E), intensity (I), polarization (θ), and phase coherence (φ) for interference puzzles [9].
- **Power & Energy Management:** Photon emissions draw from a **battery** or energy pool. High-intensity beams or "overcharge" pulses risk overheating. This introduces a risk/reward layer: you can fire stronger beams (increased range) at the cost of heat or battery, requiring careful management [11] [12].
- **Progression Systems:** Players **unlock new optics and wavelengths** over time (horizontal progression) and improve tools/stats (vertical progression) [13] [14]. For example, gaining access to UV or IR lasers opens new puzzle types, while upgrades boost beam power or detector sensitivity. Long-term progression motivates repeated runs, with *spectral tokens* earned in each run to unlock future modules [15] [14].
- **Data & Analysis UI:** A dedicated UI layer displays mission objectives, photon statistics, and feedback to reinforce learning. It includes status readouts and histograms of detected wavelengths [16] [17].

By combining these systems, Spectra's **core loop** works as follows: the player sets the beam parameters and fires photons, the Photonics engine propagates beams through the environment, optical elements transform the beams or trigger reactions, and detectors log successful deliveries towards mission goals [18] [19].

Physics-Based Mechanics

Spectra's mechanics are grounded in real optics laws. Each photon has *base properties*:
- **Wavelength (λ)** – determines its energy via $E = 1240 / \lambda$ (with λ in nm, E in eV) [20].

- **Polarization (θ)** – a beam's polarization angle; only aligned polarizers will transmit it [20].
- **Coherence (φ)** – its phase; used in interference puzzles.

As photons move, they **reflect, refract, absorb, scatter, or split** upon hitting an object [4]. Beam intensity decays over distance: $I_{out} = I_{in} \cdot \exp(-\alpha(\lambda) \cdot d)$, per the Beer–Lambert law [10] [21]. The attenuation coefficient α depends on the medium or filter. For example, passing through a colored filter might apply an exponential dimming.

Key component interactions are implemented with simple rules (see "Optical Interactions" below). For instance, mirrors reflect beams (angle-in = angle-out), prisms split beams into spectral components, polarizers attenuate by $\cos^2(\Delta\theta)$ [22], and fluorescence targets absorb and re-emit light at shifted wavelengths. Detectors count photons in target bands and log spectra [23]. This direct mapping ensures that every gameplay effect has a real optical counterpart [5].

Optical Interactions

- **Mirror:** Reflects the beam with incident angle = reflection angle [24]. Multiple bounces should be handled safely to avoid infinite loops.
- **Prism:** Splits an incoming beam into constituent wavelengths. Implementation spawns multiple photons at offset λ bands [25].
- **Polarizer:** Transmits only aligned polarization. The transmitted intensity is multiplied by $T = \cos^2(\theta_{in} - \theta_{filter})$ [22].
- **Sample (Absorber):** Material with a Gaussian absorbance curve selectively attenuates wavelengths and may **fluoresce**. A photon has an absorption probability based on its λ. If absorbed, there is a chance of delayed emission (fluorescence) at a longer wavelength. The sample's state may change (excited).
- **Detector:** Counts photons within its λ sensitivity range. Each hit increments a count and logs λ for a histogram, triggering mission progress when thresholds are met [23].
- **Filter/Grating:** Passes or deflects specific wavelengths, enabling color-tuning or gating puzzles (e.g. only red light passes).
- **Photochemical Target:** Requires a minimum photon energy ($E \geq$ threshold) to trigger. If hit with sufficient energy, it reacts (breaking bonds, emitting new photons, unlocking gates) [26].

In each frame, the **reaction logic flow** is: Photon fired → intersect optical object → evaluate interaction type → update photon's state (λ, E, I, θ, φ) → spawn any secondary emissions (fluorescence, reflections) or trigger reactions → update mission/detector stats [18].

Player Controls & Feedback

Players directly control the photon core in a first-person or top-down view. Input mappings are designed for **immediate, intuitive control** [27]:

- **Move:** WASD keys (with smooth inertial motion) [7].
- **Aim:** Mouse movement (or lock-on with right-click) rotates the beam direction [28].
- **Fire Photon:** Spacebar or left-click emits a photon packet in the aimed direction [29].
- **Tune Wavelength (λ):** Mouse wheel or keyboard scrolling changes the beam's wavelength (shifting its color) in real time [30].

- **Adjust Polarization (θ):** Keys Q/E rotate the beam's polarization axis [31].
- **Overcharge Pulse:** Holding fire (space) builds a high-intensity beam at the cost of battery/heat [12].
- **Analyze/Pause:** Tapping Tab brings up a spectrograph overlay (showing live intensity vs. wavelength) [32].

The game targets **very low input latency** (<50 ms) so actions feel instantaneous [27]. Every change (e.g. sliding λ or rotating a polarizer) produces immediate visual/audio feedback: the beam's color hue and audio pitch change with λ, and intensity changes show as beam brightness or opacity [33] [27]. Subtle cues (a click or flash when a detector triggers, a glow on sample excitation) reinforce interaction. This continuous, unbroken feedback loop ensures the player always sees the effect of their actions [27] [34].

The HUD is minimalist but data-rich [35] [36]. Key elements include:

- **Top-Left:** Photon status (current λ, energy E, polarization, battery/heat).
- **Bottom-Left:** Power and heat meters.
- **Top-Right:** Mission panel (objectives, detectors active).
- **Center-Right:** Mini-spectrograph (histogram of wavelengths hitting detectors) [17].
- **Bottom-Right:** Contextual tooltips (e.g. material properties on hover).

Visual design ties color to data: hue encodes wavelength, brightness encodes intensity [35]. UI highlights critical info (active wavelength band, battery level) with size/color and uses intuitive icons (e.g. prism for diffraction) [36]. Tooltips explain physics concepts as needed. The interface maintains a consistent palette and avoids clutter, letting players focus on experimentation [36].

Progression & Upgrades

Spectra's progression interleaves **unlocking new optics/wavelengths** (horizontal progression) and **boosting capabilities** (vertical progression) [14] [37]. Early levels might allow only visible light; solving puzzles earns tokens to unlock ultraviolet (UV) or infrared (IR) sources, effectively giving players new "spells" for experiments [14]. For example, UV (200–400 nm) unlocks photochemical bond-breaking reactions [38], while IR (700–1000 nm) enables thermal effects like heating objects [39]. Similarly, upgrades increase stats: stronger lasers (more beam intensity/pulse duration) and more sensitive detectors mimic "leveling up" a scientist's tools [37] [40].

Concretely, planned unlock tiers [41] [42] : - **Tier 1 – Visible (600–700 nm):** Basic reflection/detection puzzles.
- **Tier 2 – Full Visible (400–700 nm):** Enables color-coded chemistry.
- **Tier 3 – UV (200–400 nm):** Triggers photochemical reactions (bond breaking).
- **Tier 4 – IR (700–1000 nm):** Introduces heat manipulation puzzles.
- **Tier 5 – Coherence Control:** Allows interference-based puzzles (wave effects).
- **Tier 6 – Polarization Layer:** Adds polarized optics mechanics.

Upgrade categories [43] include *Source Power* (beam strength vs. heat), *Optical Precision* (beam focus), *Detector Sensitivity* (lower noise), *Cooling Systems* (heat tolerance), and *Spectral Expansion* (unlock λ bands). Players select upgrades in a modular tree, so playstyle can lean towards brute force (max power) or finesse (better focus/sensitivity).

Progression loops are built like a light-themes roguelike [44] : each calibration **run** through the instrument is one experiment. Completing a module yields spectral tokens to spend on unlocks for the next run [15] . Over multiple runs, the player gradually restores the full spectrometer, gaining access to deeper challenges. This loop reinforces that solving light-based puzzles *teaches* new optics, which *unlocks* new puzzles [44] [15] .

Level & Environmental Design

Levels are themed optical chambers with special zones (see table below):

- **Optical Chambers:** Puzzle rooms built from closed mirror/prism circuits – classic laser-maze challenges [45] .
- **Reactive Zones:** Photochemical arenas; e.g. a pool of dye that reconfigures its layout when illuminated [46] .
- **Dark Matter Fields:** Regions that **absorb all light**, forcing players to route beams around or use spectral tricks (e.g. IR heating to make other paths). These create natural detours requiring spectral adaptation [47] .
- **Detector Banks:** Goal zones with multiple detectors; success is measured by delivering specific wavelength mixes [48] .
- **Cooling Flow Paths:** Thermal puzzles where players must route lasers to dissipate heat away, akin to fluid flow puzzles [49] .

Each environment may combine these: for example, a chamber could have a hot zone (IR needed) and also a polarization puzzle, requiring sequential solution of sub-mechanics.

Technical Architecture

Spectra's code is organized into clear subsystems [50] :

- **PhotonEngine:** Manages photon entities. Handles emission, straight-line propagation, and attenuation. Applies physics laws (reflection, refraction, polarization changes) as photons interact.
- **OpticsSystem:** Contains all optical elements (mirrors, samples, etc.) and resolves collisions. On intersect, it applies the rules from "Optical Interactions" and spawns any secondary photons (reflections, fluorescence).
- **SpectroscopySystem:** Collects data: it tracks how many photons hit each detector and maintains a spectrum histogram. It also evaluates triggers (e.g. if enough energy has been delivered).
- **ProgressionManager:** Manages unlocks, upgrades, and the meta-progression loop (awarding tokens post-run).
- **UIManager:** Updates HUD elements, spectrograph overlays, and tooltips in real time.

Data for game entities (samples, detectors, etc.) are defined via JSON or scriptable data. For example, a sample might be defined by its absorbance curve and fluorescence probability [51] . This data-driven approach lets designers tweak physics without changing code.

System Flow (Photon Lifecycle)

1. **Emit Photon:** When the player fires, PhotonEngine creates a new Photon object with attributes {position, direction, wavelength λ, intensity I, polarization θ, phase φ}. (Energy E is computed as E=1240/λ [20].)
2. **Propagate:** In each update, the engine advances the photon until it collides or leaves the chamber. Intensity decays with distance: $I = \exp(-\alpha \cdot dx)$* [10].
3. **Interact:** Upon hitting an object, PhotonEngine delegates to OpticsSystem:
4. For a **mirror**, compute reflection direction.
5. For a **prism**, delete the incoming photon and spawn new photons of split wavelengths.
6. For a **polarizer**, multiply intensity by $\cos^2(\Delta\theta)$ [22] (if absorption, photon may be destroyed).
7. For a **sample**, compute absorption: use a Gaussian curve to get absorption chance. If absorbed, optionally trigger fluorescence (spawn a new photon at shifted λ).
8. For a **detector**, if wavelength ∈ target band, increment its count and possibly destroy the photon [23].
9. **Physics Update:** After each hit, update photon's state (new λ, I, θ). If the photon's intensity falls below a threshold or it is absorbed, remove it.
10. **Feedback/Events:** SpectroscopySystem updates histograms on detector hits. If detectors meet mission criteria, flag level completion. If photochemical thresholds are reached (E ≥ material threshold [52]), trigger reaction events (color change, unlock gate, etc.).

Pseudocode Examples

Below are simplified pseudocode snippets for core mechanics:

```
// Emit a new photon from the source
function firePhoton(source):
    photon = new Photon()
    photon.position = source.position
    photon.direction = source.aimVector
    photon.wavelength = source.currentWavelength
    photon.energy = 1240.0 / photon.wavelength   // E = 1240/λ eV [20]
    photon.intensity = source.beamIntensity
    photon.polarization = source.polarizationAngle
    photon.phase = 0
    return photon

// Main loop: propagate photons and handle interactions
function updatePhoton(photon, deltaTime):
    while photon.exists:
        hit = findNearestIntersection(photon, deltaTime)
        if not hit:
            break
```

```
        // Move photon to hit point
        travelDistance = distance(photon.position, hit.position)
        photon.intensity *= exp(-hit.material.alpha * travelDistance)
        photon.position = hit.position
        // Handle hit object
        handleInteraction(photon, hit.object)
        if photon.intensity < MIN_INTENSITY:
            photon.exists = false
```

```
// Handle beam interaction with an object
function handleInteraction(photon, object):
    if object.type == MIRROR:
        photon.direction = reflect(photon.direction, object.normal)  // θ_out =
θ_in [24]
    elif object.type == PRISM:
        for each colorBand in object.splittingBands:
            newPhoton = clone(photon)
            newPhoton.wavelength = colorBand.wavelength
            newPhoton.direction = computeRefractDirection(photon.direction,
colorBand)
            queuePhoton(newPhoton)  // spawn new photons [25]
        photon.exists = false
    elif object.type == POLARIZER:
        alignment = cos(photon.polarization - object.axisAngle)
        photon.intensity *= alignment * alignment           // Malus's law [22]
        // if not aligned, photon may be destroyed by low intensity
    elif object.type == SAMPLE:
        absorbProb = sampleAbsorbance(object.absorbCurve, photon.wavelength)
        if random() < absorbProb:
            photon.exists = false     // photon absorbed
            // possible fluorescence
            if random() < object.fluorescence.prob:
                spawnFluorescence(object, photon.wavelength)
    elif object.type == DETECTOR:
        if object.range.contains(photon.wavelength):
            object.count += 1
            recordSpectrum(photon.wavelength)
        photon.exists = false
    // Add other object types as needed
```

```
// Example: spawn a fluorescence photon
function spawnFluorescence(sample, originalWavelength):
    newPhoton = new Photon()
    newPhoton.position = sample.position
    newPhoton.direction = randomDirection()
```

```
    newPhoton.wavelength = originalWavelength + sample.fluorescence.shift
    newPhoton.energy = 1240.0 / newPhoton.wavelength
    newPhoton.intensity = photon.intensity * sample.fluorescence.prob
    queuePhoton(newPhoton)
```

These code snippets illustrate the **photon lifecycle**: firing, propagating with decay, interacting via physics rules, and updating detectors.

Development Roadmap

To build Spectra efficiently, we prioritize foundational systems first:

1. **Photon Physics Engine:** Prototype and finalize beam mechanics (reflection, refraction, attenuation, polarization, fluorescence) [53] . This core must be accurate to form gameplay's backbone.
2. **Optical & Detector Classes:** Implement data-driven objects for samples, filters, detectors, etc., with event hooks for interactions [54] . Early creation of sample/detector classes allows testing of physics.
3. **Controls & Input System:** Develop smooth, low-latency controls with real-time tuning (λ/ polarization) [34] [27] . Ensure input feedback is immediate (<50 ms) for 'game feel' [27] .
4. **UI Overlay Prototype:** Build the HUD and spectrograph UI elements (wavelength slider, battery meter, intensity histogram) [54] . A working UI helps visualize and debug photon mechanics.
5. **Mission Loop Prototype:** Assemble one full puzzle: Source → Sample → Detector → Analysis → Unlock [55] . This end-to-end test validates game flow and progression token rewards.
6. **Meta-Progression & Save System:** Integrate saving of upgrades/unlocks between runs [55] . Ensure players' horizontal/vertical progression is persistent.
7. **Automated Testing:** Develop a suite to verify wavelength–energy consistency and physics accuracy (e.g. test that E=1240/λ, Beer–Lambert effects match expectations) [56] . This guards against regression in complex optics code.

Each stage builds on the previous. First, ensure photons behave correctly in an empty level; then add objects and test interactions; then wrap with player controls and UI; finally connect it all in a playable loop. This plan follows the key priorities outlined in the design draft [19] .

System Architecture (Textual Diagram)

- **Player Entity:** Has components *PhotonEmitter* (with tunable λ, θ, power, battery, overcharge pulse) and *MovementController*.
- **PhotonEngine:** Central loop that iterates all active Photon entities. For each photon, calls `propagate()` (physics) and dispatches to OpticsSystem on hit.
- **OpticsSystem:** Contains lists of scene objects. On photon-hit, it executes the rules above, possibly invoking SpectroscopySystem (for detectors) or spawning new photons (for prisms/fluorescence).
- **SpectroscopySystem:** Maintains detectors' data. Listens for "photon detected" events and updates mission state/UI. Also computes histograms for the UIManager to draw.
- **ProgressionManager:** Manages unlocked features (wavelengths, upgrades). Consumes in-game currency (spectral tokens) to enable new items in the next session.
- **UIManager:** Observes game state (photon emitter, detectors, mission) and updates HUD widgets. Also handles animated feedback (flashes, tooltips).

These subsystems communicate via defined interfaces or event messages. For example, when a photon intersects a detector, OpticsSystem emits a “PhotonDetected(wavelength)” event that the SpectroscopySystem and UIManager both listen to. The architecture is component-based and modular, facilitating independent testing of PhotonEngine vs. UI vs. progression.

Summary

This blueprint unifies real optical physics with structured game design. By using authentic laws (Beer–Lambert decay, Malus’s law, energy thresholds) tied to deliberate mechanics, Spectra will deliver puzzles that are both **educational and engaging** [5] [57]. The developer-focused sections above break down every system—player controls, physics engine, progression, UI—so implementation can proceed systematically from concept to code. The prioritized roadmap ensures core gameplay is playable early, and the detailed pseudocode and flow logic give clear guidance for coding the critical photon lifecycle.

Sources: Principles and examples from optics-based games [58] [57], game feel and UI research [27] [36] [57], and the original Spectra design document [4] [59] have been integrated to create this comprehensive design foundation.

---

[1] [2] [3] [4] [6] [7] [8] [9] [10] [11] [12] [13] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [28] [29] [30] [31] [32] [33] [34] [35] [38] [39] [40] [41] [42] [43] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [59] Spectra Game Core Systems Blueprint design 01.docx
file://file-FtK88cpARsv7cvWer5ypZm

[5] [14] [27] [36] [37] [44] [57] [58] Light-Based Game Mechanics.pdf
file://file-49QgWCibyL7U71eMhffhKV