# Technical Report

Weaver of Light and Shadow

# Introduction

This project uses light as the core mechanic in a first person puzzle action game. The goal is to design a game where illumination directly influences navigation, combat, and player decision-making, while also demonstrating ability in Unreal Engine 5's C++ gameplay framework.

# Technical Stack

## MyShadowBridge

### Design

The ShadowBridge system was designed to make **light a functional gameplay mechanic**, rather than a purely visual element. The bridge responds dynamically to illumination in the environment, allowing players to unlock traversal paths through intentional light placement and spell usage.

Key design goals included:

- Treat light as a **gameplay input**
- Support **multiple light sources** (torches and Lumos)
- Enable activation based on **accumulated illumination**, not a binary trigger
- Provide reliable activation even when light only partially covers the bridge

### System Architecture

The system is implemented in a dedicated C++ actor, `AMyShadowBridge`, which encapsulates all logic related to light sampling and bridge activation.

Responsibilities of `AMyShadowBridge` include:

- Owning and managing the bridge mesh
- Controlling **visibility and collision** based on lighting state
- Running a **timer-based sampling loop** to evaluate lighting conditions
- Aggregating light contributions from both `UPointLightComponent` (Torches) and `USpotLightComponent` (Lumos)

## Light Sampling Strategy

Instead of sampling illumination at a single point, the bridge defines **multiple sample points in local space** along its length.

Each lighting check performs the following steps:

1. Transform each local sample point into world space using the bridge's component transform
2. Evaluate accumulated light intensity at each world-space sample
3. Track the maximum intensity observed across all samples

Using multiple sample points avoids several common issues:

- **Single-point false negatives**, where part of the bridge is lit but the sampled point is not
- Edge cases where the player illuminates only one end of the bridge
- Inconsistent activation when the player stands directly on the bridge

## Intensity Accumulation Algorithm

For each sample point, the total light intensity is computed as the **sum of all contributing point lights and spotlights:**

```
AccumulatedIntensity = Σ(PointLightContribution) + Σ(SpotLightContribution)
```

The bridge activates when the maximum sampled intensity exceeds a configurable threshold.

### Point Light Contribution(Torches)

Point lights use a **distance-based quadratic falloff model**:

- Lights outside the attenuation radius contribute zero intensity
- Intensity smoothly decreases with distance

Formula:

```
t = 1 - (distance / attenuationRadius)
intensity = baseIntensity × t²
```

### Spot Light Contribution(Lumos)

Spot lights combine **distance falloff** and **angular falloff**:

1. Distance falloff is computed using the same quadratic model as point lights
2. Angular falloff is computed using the dot product between:
    - The spotlight's forward direction
    - The direction from the light to the sample point

Steps:

- Compute the dot product
- Convert the result to an angle in degrees
- Linearly interpolate intensity between the inner and outer cone angles

## Performance Consideration

To avoid unnecessary per-frame computation, the bridge uses a **timer-based sampling system** instead of `Tick()`:

- Sampling frequency is controlled via `CheckInterval`
- Early-exit logic stops sampling once the activation threshold is reached
- Lights that are invisible or do not affect the world are ignored
- The system remains scalable even with many lights present in the scene

## Challenges and solution

**Challenge 1: Choosing Between Binary Triggers and Physical Light Evaluation**
**Problem:**
A simple overlap trigger or binary switch would activate the shadow bridges easily and cannot custom bridge instances.

**Solution:**
The system evaluates **accumulated light intensity** using physically inspired falloff models for both point and spot lights. This allows the system to be more scalable and some of the shadow

bridges need to be set with a high threshold to ensure it can be only activated by wand spell Lumos.

### Challenge 2: Designing a Gameplay-Friendly Light Falloff Model

**Problem:**
Using Unreal Engine's rendered lighting data directly is not suitable for gameplay logic, as it is view-dependent and expensive to query. Additionally, simple linear falloff produced abrupt activation changes that felt unintuitive during playtesting.

**Solution:**
The system adopts **physically inspired light attenuation models commonly used in real-time rendering**, simplified and adapted for gameplay use.

- **Point lights** use a quadratic distance falloff based on normalized attenuation radius, inspired by standard inverse-square attenuation models used in game engines.
- **Spot lights** combine distance-based attenuation with angular falloff derived from dot-product cone evaluation, consistent with conventional spotlight shading equations.

### Challenge 3: Inconsistent Activation When Player Stands on the Bridge

**Problem:**
When the player stands on the bridge, their body can block Lumos from reaching a single sampling point, causing the bridge to deactivate unexpectedly.

**Solution:**
Multiple sampling points distributed along the bridge ensure that light is detected even if some probes are occluded, maintaining stable activation during traversal.

# MyGhost

## Design

The Ghost enemy was designed to serve as a **threat** that reinforces careful movement and spatial awareness rather than complex combat mechanics. The primary goals of the system were:

- Provide **clear and readable enemy behavior** that players can learn and predict.
- Support **patrol and chase states** driven by player proximity.
- Enforce **deterministic instant-kill interactions** to raise tension without relying on combat depth.
- Operate independently of Unreal's built-in AI framework to maintain **full control over behavior logic** and reduce overhead.

## System Architecture

The Ghost AI is implemented entirely in the `AMyGhost` C++ actor and is composed of the following subsystems:

- **State Machine**
  Manages transitions between Patrol and Chase states.
- **Movement Controller**
  Handles horizontal navigation toward patrol targets or the player using vector-based steering.
- **Hover & Gravity Simulation**
  Adds visual motion and ensures stable interaction with walkable surfaces.
- **Player Interaction System**
  Uses collision-based overlap detection to trigger instant player death.
- **Audio Feedback**
  Provides movement and state-change cues to communicate threat escalation.

### AI State Machine

The Ghost operates using a simple two-state finite state machine:

```
enum class EGhostState

{

    Patrol,

    Chase

};
```

### Patrol State

- The Ghost moves back and forth between:
    - Its spawn location (`StartLoc`)
    - A target offset defined by `WanderOffset`
- Movement direction switches when the Ghost reaches a configurable distance threshold.
- This creates predictable back-and-forth motion suitable for environmental puzzles.

### Chase State

- Triggered when the player enters `DetectRadius`.
- The Ghost directly moves toward the player's XY position.
- Chase speed is increased relative to patrol speed.
- The Ghost exits chase mode when the player exceeds `LoseRadius`.

### Transition Logic

State transitions are evaluated every tick using **distance-based checks** in the horizontal plane:

```
float Dist = FVector::Dist(GhostXY, PlayerXY);
```

## Movement Algorithm

The Ghost uses **manual vector-based movement** rather than Unreal's AI pathfinding:

1. Compute direction vector toward the destination.
2. Normalize direction and scale by speed and `DeltaTime`.
3. Clamp movement to avoid overshooting the target.
4. Apply rotation so the Ghost visually faces its movement direction.

## Hover and Ground Handling

To create a floating, spectral effect, the Ghost combines **hover motion** with **ground detection**.

### Hover Motion

- Uses a sinusoidal function to offset the Ghost's vertical position:
  `offset = sin(time × speed) × amplitude`
- Applied only when the Ghost is standing on a walkable surface.

### Ground Detection

- A downward line trace checks for actors tagged as `"Walkable"`.
- If grounded, the Ghost:
  - Snaps slightly above the surface
  - Resets vertical velocity
- If not grounded, gravity is applied manually.

## Player Interaction & Kill Logic

The Ghost enforces **instant-kill behavior** via collision overlap:

- A `USphereComponent` (`HurtTrigger`) is configured to overlap only with pawns.
- On overlap:
  - The actor is validated as the player
  - A death function is triggered immediately

```
if (OtherActor->ActorHasTag("Player"))
{
    KillPlayer(OtherActor);
}
```

## Audio Feedback Integration

### Movement Sound

- Played at intervals while the Ghost is moving.
- Interval decreases during Chase to increase perceived urgency.

**Chase Sound**

- Triggered once when entering Chase state.
- Reinforces threat escalation without UI indicators.

## Challenges and Solutions

### Challenge 1: Challenge 1: Ghost Interacting with Unintended Actors

**Problem:**

During chase behavior, the Ghost shares the same physical space as other gameplay actors, including ShadowBridges. Without strict filtering, overlap events could be triggered on unintended actors, leading to incorrect behavior such as the Ghost destroying or interacting with non-player objects.

**Solution:**

A tag-based filtering approach was introduced to strictly constrain kill logic to the player only. The Ghost verifies that the overlapping actor both:

- Is a Pawn
- Possesses the "Player" gameplay tag

Only when both conditions are met does the Ghost invoke the player death routine. This guarantees deterministic instant-kill behavior while preventing unintended interactions with environmental actors such as ShadowBridges.

### Challenge 2: KillZ, gravity and Ground detection

**Problem:**

The Ghost must be able to patrol and chase across valid surfaces (floors and active ShadowBridges), while also falling and being destroyed when no longer supported. Relying solely on collision or KillZ checks was insufficient, as it could result in floating behavior, premature destruction, or incorrect grounding on inactive bridges.

**Solution:**

A hybrid movement and grounding system was implemented combining:

- Explicit ground detection using a downward line trace that only recognizes actors tagged as "Walkable" (including active ShadowBridges).
- Manual gravity simulation when no valid ground is detected, allowing the Ghost to fall naturally.
- KillZ validation as a final safety net to destroy the Ghost if it falls below the world's configured vertical bounds.

**This layered approach ensures the Ghost:**

- Moves reliably on valid traversal surfaces
- Falls correctly when a ShadowBridge deactivates
- Is safely cleaned up when exiting the playable space

# MyWand

## Design Goals

The Wand system serves as the player's **interaction tool**, responsible for manipulating light, combat, and environmental puzzles. The design goals were to:

- Centralize all spell-related gameplay logic.
- Support **multiple interaction types** (torch toggling, enemy elimination, Lumos casting).
- Introduce a **charge-based resource system** to constrain player actions.
- Decouple input handling from gameplay execution to ensure consistency across levels.
- Provide deterministic, raycast-based interactions aligned with player camera direction.

## System Architecture

- **Static Mesh Component**
    - Represents the visual wand model.
    - Collision is disabled to prevent interference with player movement.
- **Spot Light Component (Lumos)**
    - Used to emit a strong, directional light.

    - Tunable parameters control intensity, cone angles, color, and shadow casting.
- **Audio Integration**
    - Plays contextual sound effects for attacks and Lumos activation.
- **Charge Management Subsystem**
    - Tracks remaining spell charges.
    - Enforces resource constraints for offensive and utility spells.

## Interaction Model

### Raycast-Based Target Detection

All wand interactions rely on a **camera-aligned line trace** to determine the targeted actor:

1. Retrieve the player camera's world position and rotation.
2. Compute a forward direction vector.
3. Perform a visibility line trace up to a configurable maximum distance.
4. Ignore the wand itself and its owning character.

### Torch Interaction

- The wand performs a raycast to identify the targeted actor.
- If the actor is of type `ATorch`, the wand toggles its light state.
- No charge is consumed.

### Enemy Attack Logic

1. Identify the aimed actor via raycast.
2. Verify the actor is tagged as `"Enemy"`.
3. Attempt to consume a charge.
4. Play attack sound effect.
5. Destroy the enemy actor.

### Lumos (Strong Light)

Lumos emits a powerful, directional spotlight used primarily to activate ShadowBridges and solve light-based puzzles.

- Lumos is implemented using a `USpotLightComponent`.
- When activated:
    - A charge is consumed.
    - Light parameters are applied.
    - The light becomes visible.
    - A timer is started to automatically deactivate the light after a fixed duration.

## Charge Management

Charges act as a **shared resource** across offensive and utility spells, forcing players to make strategic decisions rather than spam abilities.

### Implementation

- Charges are tracked as an integer counter (`ChargeCount`).
- A helper method (`ConsumeCharge`) decrements charges atomically.
- Charge availability is checked before executing any charge-consuming action.

## Audio Feedback Integration

#### Attack Sound
Played when an enemy is successfully eliminated.

#### Lumos Sound

Played upon Lumos activation.

## Challenges and Solutions

### Challenge 1: Preventing Unintended Charge Consumption

**Problem:**
Early implementations consumed charges before validating targets, allowing players to lose charges when attacking non-enemy actors.

**Solution:**
Target validation was reordered to ensure charges are only consumed **after** confirming a valid enemy target. This guarantees fair and predictable resource usage.

### Challenge 2: Connecting the spells with the player's input

**Problem:**
Custom Enhanced Input Mapping Contexts were initially used to bind spell actions. However, during multi-level gameplay, these custom contexts failed to activate consistently, resulting in spells not triggering or becoming unresponsive after level transitions.

**Solution:**
Spell input actions were rebound to the Default Input Mapping Context, which is guaranteed to be active across all levels and persists through level transitions.

# Torch

## Design Goals

The Torch system provides a **static, persistent light source** that players can interact with using the wand. Torches serve as the foundational light mechanic in the game, enabling environmental interaction and contributing to light-based puzzle solutions such as activating Shadow Bridges.

- Provide a **stable light source**
- Allow **player-controlled interaction** without resource cost
- Integrate seamlessly with the ShadowBridge lighting system

## System Architecture

- Torch mesh and point light component
- Tracks whether the torch is currently lit
- Toggles visibility and lighting state
- Plays appropriate audio feedback on state change

## Interaction Logic

Torch interaction is driven by the wand's raycast-based targeting system:

1. The wand detects an aimed actor
2. If the actor is an `ATorch`, it calls `ToggleLight()`
3. The torch switches its internal `bIsLit` state
4. Light visibility and audio feedback are updated accordingly

### Lighting Configuration

Torches use **PointLightComponents** with physically inspired parameters:

- Editable attenuation radius and intensity exposed to Blueprint
- Warm color temperature

### Audio Feedback

Each torch provides immediate audio feedback:

- One sound for lighting up
- One sound for extinguishing

## WeaverOfLightNShadowCharacter

### Purpose and Role

AWeaverOfLightNShadowCharacter is Unreal Engine's default first-person character framework and I implemented some custom gameplay systems in this project. While the base class provides locomotion, camera control, and physics, this module extends the character with:

- Wand-based spell interaction (Torch toggle, Attack, Lumos)
- Player death and respawn management
- Audio feedback for movement, jumping, and death
- Runtime safety checks (KillZ handling)

### Input Handling and Wand Integration

- Input is handled using UE5 Enhanced Input

- Input actions (ToggleTorch, Attack, Lumos) are bound in SetupPlayerInputComponent
- Each handler (HandleToggleTorch, HandleAttack, HandleLumos) retrieves the currently attached AMyWand actor via GetWand()
- The character does not implement spell logic directly, but forwards intent to the wand

```
if (AMyWand* Wand = GetWand())
{
    Wand->ActivateStrongLight();
}
```

Rather than hard-referencing the wand in the character header, the system dynamically searches for an attached AMyWand actor at runtime:

```
GetAttachedActors(AttachedActors);
```

## Player Death Handling and KillZ Logic

Create a robust and deterministic death system that:

- Plays audio feedback before respawn
- Safely reloads the current level

**Implementation**

- CheckKillZ() runs every tick
- Uses a manual Z threshold rather than engine KillZ to allow tuning
- Guards against repeated execution using bIsDead and bDeathTimerStarted
- Plays death sound immediately
- Reloads the level after a short delay using a timer

```
GetWorldTimerManager().SetTimer(
    DeathTimerHandle,
    this,
    &AWeaverOfLightNShadowCharacter::Die,
```

```
    1.0f,
    false
);
```

## Audio System

### Footstep Audio

Footstep sounds are triggered dynamically based on movement state rather than animation events. Each frame:

- The system checks whether the character is grounded
- Movement speed is evaluated
- A timer enforces a configurable playback interval

### Jump and Landing Audio

Jump and landing audio are separated to improve clarity:

- Jump initiation remains silent
- Landing plays a distinct sound via the overridden Landed() callback

This ensures jump audio aligns with physical impact rather than input timing.

# Blueprint

## Door-to-Next-Level Blueprint

Level transitions are handled through a Blueprint-based door actor that:

- Detects player overlap
- Loads the next level using a predefined level name

Other Blueprint classes primarily serve as configuration layers for C++ systems and do not introduce additional gameplay logic.

## Materials and Textures

### Asset Pipeline

All textures used in the project were sourced from Fab, ensuring consistent visual quality and physically based rendering standards.

A reusable Master Material was authored to expose key parameters:

- Base color textures
- Emissive intensity
- Opacity
- Roughness

Material Instances were created per asset.

# Learning Outcomes

Through the development of Weaver of Light and Shadow, I gained hands-on experience designing and implementing a gameplay-driven systems architecture in Unreal Engine 5 using C++.

Key learning outcomes include:

**Systems-Oriented Gameplay Design**

I learned how to translate an abstract design concept: light as a mechanical resource into concrete gameplay systems that affect traversal, enemy behavior, and player decision-making.

**Advanced C++ Gameplay Architecture in UE5**

I developed multiple custom gameplay actors (ShadowBridge, Ghost, Wand) using Unreal's component-based architecture, timers, and collision systems. This deepened my understanding of how UE5 manages world state, actor lifecycles, and cross-system communication.

**Custom AI Logic Without Behavior Trees**

By implementing a state-driven Ghost AI entirely in C++, I learned how to construct deterministic AI behavior using distance checks, state transitions, and controlled movement logic without relying on Blueprint Behavior Trees or NavMesh systems.

**Light-Based Interaction and Spatial Reasoning**
Designing the ShadowBridge required learning how to reason about spatial sampling, light attenuation, angular falloff, and performance tradeoffs. This strengthened my ability to design approximate but robust physical models suitable for real-time gameplay.

**Robust Input and State Management**
Integrating Enhanced Input with C++ gameplay logic taught me how to decouple input intent from action execution, ensuring consistent behavior across level transitions and builds.

**Debugging and Iterative Problem Solving**
Throughout development, I encountered and resolved issues involving collision filtering, unintended actor interactions, KillZ handling, and runtime input failures. These challenges improved my ability to diagnose engine-level issues and implement clean, maintainable fixes.

# Future Work

While the current implementation achieves the core gameplay goals, several extensions could further expand the depth, polish, and scalability of the project:

**Animation-Driven Feedback**
Integrating character and enemy animations (attack, casting, death) with animation notifies would allow tighter synchronization between visuals, audio, and gameplay logic.

**Expanded Light Interaction Mechanics**
Future iterations could introduce additional light types, such as moving light sources, reflective surfaces, or temporary light projectiles to increase puzzle complexity and emergent gameplay.

**Behavior Tree–Based AI Extension**
While the current Ghost AI is fully deterministic, migrating or augmenting it with Behavior Trees could enable more complex behaviors such as flanking, retreating, or group coordination.

**Performance Optimization at Scale**

For larger environments, the light sampling system could be optimized further using spatial partitioning or selective light queries to reduce per-frame or per-timer computation costs.

**Narrative and Environmental Storytelling**

Environmental cues, visual storytelling, and light-based narrative elements could be layered onto the existing systems to enhance player immersion without altering core mechanics.

**Accessibility and UX Enhancements**

Adding visual indicators for charge availability, light intensity feedback, and clearer tutorial prompts would improve readability and accessibility for a broader range of players.