

Absolutely! I can help you craft a professional-level production document step by step for your 2D platformer or RPG project. Since the instructions are very structured, I'll create a complete template with content that fits your word counts and requirements. You can then tweak it to match your actual project. I'll assume a sample project called "Eclipse Runner", a 2D platformer for PC.

Eclipse Runner – Production Document

Step 1 — Project Overview

Project Name: Eclipse Runner

Genre: 2D Platformer

Target Platform and Audience: PC; casual and mid-core players aged 12+ who enjoy action-platforming and exploration.

Team Size and Roles: 3 developers — Alex (Programmer & Game Designer), Jordan (Artist & UI Designer), Sam (Producer & Level Designer)

Duration and Milestones: 8-week development cycle; Week 1–2: Concept and prototyping, Week 3–5: Core mechanics and level creation, Week 6–7: Polish and testing, Week 8: Final build and documentation.

Project Goal and Vision:

Eclipse Runner is a 2D platformer that challenges players with fast-paced platforming, combat, and collectible-based progression across 4 unique levels. The vision was to create a polished prototype demonstrating core platforming mechanics, engaging combat, and level design principles suitable for PC players. The project aimed to teach both the team and potential players about iterative design, balanced difficulty, and integrating feedback from early playtests. With limited art assets, placeholder sprites were used to focus on gameplay and systems rather than aesthetics. The ultimate goal was a playable, cohesive demo that could serve as a foundation for future expansion or a full game.

Step 2 — Development Report

Tools and Engine:

Engine: Unity 2022.3 LTS

Programming Language: C#

Art: Aseprite for pixel sprites

Sound: Bfxr for placeholder effects, Audacity for editing

Plugins: Cinemachine (camera control), DOTween (animations), ProBuilder (level prototyping)

Workflow:

Development began with rapid prototyping of core mechanics: player movement, jumping, and combat. Initial prototypes were built in a minimal "test arena" scene. Feedback from the team and iterative testing informed adjustments to physics, jump arc, and combat hitboxes. Once core mechanics were stable, level design began using modular tilesets in ProBuilder, gradually expanding from simple layouts to fully playable 4-level zones.

Major Technical and Design Decisions:

Collision system relied on Unity's 2D physics with custom edge-correction scripts to prevent clipping.

Enemy AI was initially simple patrols but evolved to include basic attack patterns to increase challenge.

Collectibles were designed to reinforce progression, with a small XP system unlocking higher jumps and dash abilities.

Scope Management:

Early scope included 6 levels and multiple enemy types. Mid-development, scope was reduced to 4 levels and 3 enemy types to meet the 8-week timeline. Additional cosmetic effects were deprioritized in favor of polishing mechanics.

Mid-Development Changes:

Player dash mechanic added after playtests revealed levels felt slow.

Combat hit detection refined to prevent unfair enemy hits.

Level 3 rebalanced after initial testing showed pacing bottlenecks.

Workflow Diagram (Conceptual):

Concept → Prototype → Playtest → Iteration → Level Build → Polish → QA → Release

Step 3 — Problems Encountered

Collision Detection Issues: Player sometimes clipped through platforms due to Unity's physics engine limitations with fast movement. Caused level frustration during playtests.

Enemy AI Complexity: Patrols and attack patterns caused performance drops when multiple enemies existed in the same scene.

Level Flow Imbalance: Level 3 was too long and repetitive; players lost engagement before reaching the boss area.

Time Constraints: With a fixed 8-week schedule, some mechanics like collectibles combo scoring had to be cut.

Art Asset Limitations: Placeholder sprites made it difficult to judge readability of enemies and items in gameplay, sometimes confusing testers.

Step 4 — Solutions & Adjustments

Collision Fix: Implemented custom edge-correction scripts and adjusted Rigidbody2D settings to reduce clipping.

AI Optimization: Simplified enemy logic to fewer state checks per frame; implemented pooling to reduce object instantiation overhead.

Level Redesign: Shortened Level 3, added vertical platforming sections, and introduced new hazards to maintain player interest.

Scope Management: Prioritized core mechanics; cosmetic features and advanced combos postponed for future updates.

Asset Readability: Adjusted color contrast, added simple outlines, and used particle effects to differentiate collectibles and enemies clearly.

Step 5 — Post-Mortem Analysis

Successes:

Core mechanics (movement, jump, dash, combat) felt responsive and engaging.

Unity engine and DOTween plugin enabled rapid iteration on animations and physics.

Team collaboration was smooth; weekly milestone check-ins ensured alignment.

Iterative playtesting helped identify and correct pacing and difficulty issues early.

Failures:

Some planned features were cut due to time constraints.

Placeholder art limited visual clarity, affecting player feedback.

Minor bugs persisted in collision edge cases, particularly at high speeds.

Root Causes:

Overambitious initial scope caused stress on timeline and feature completion.

Limited asset planning slowed polish and playtesting readability.

Lack of early AI optimization led to late-stage performance issues.

Step 6 — Key Learnings

Technical: Early testing of physics interactions prevents late-stage clipping issues; object pooling improves performance with multiple AI entities.

Design: Gradual introduction of mechanics (e.g., dash, attack) keeps players engaged; balancing level flow early prevents pacing issues.

Production / Team: Defining realistic scope and prioritizing core features ensures project completion; regular milestone check-ins improve collaboration.

Player Engagement: Collectible placement should reward exploration without disrupting level pacing.

Documentation: Maintaining structured development notes allowed the team to track decisions, identify recurring problems, and streamline post-mortem analysis.