System Verification and Validation Plan for SFWRENG 4G06 - Capstone Design Project

Team #7, Wardens of the Wild
Felix Hurst
Marcos Hernandez-Rivero
BoWen Liu
Andy Liang

January 11, 2026

Revision History

Date	Version
All	1.0
Marcos Hernandez-Rivero	1.1

Contents

List of Tables

1 Symbols, Abbreviations, and Acronyms

symbol description

T Test

This document will cover our team's plan to ensure our project will do what we intend it to do, and will meet our stakeholders' needs. This will cover background information regarding our software project and team, high–level plans for verifying all of the important artifacts of our project, detailed descriptions of system and unit tests, and lastly, a list of usability survey questions.

2 General Information

This section covers key background information related to the project.

2.1 Summary

The software being tested is our game, Ad Natura. This 2D pixel art styled puzzle platformer will take a player on a short adventure through a few levels with environmental puzzles to solve. Two key functions that will be the subject of much verification and validation testing will be the function of destroying and decomposing terrain into small pixel chunks, and the function of the slime mold organism traversing through the area and reacting to environmental stimuli.

2.2 Objectives

- This project's key objectives include:

Establish confidence in software correctness for the primary two functions: Terrain decomposition and the slime mold organism.

Establish confidence in software safety and security.

Demonstrate our message of cooperation between humans and nature, through the use of aesthetics and environmental storytelling.

Demonstrate satisfactory usability for both gamers and non–gamers, to expand to a wide audience.

- Objectives that are out of scope for this project include:

Establishing confidence in software correctness for fundamental Unity engine code and functions. This is not necessary as we can assume that Unity's engine has been thoroughly verified.

Demonstrating eye–catching, memorable visuals and music. These aspects are only important for marketing purposes, which is a very late–stage concern, and therefore not within the scope of this project at this stage.

2.3 Extras

The extras include a Design Thinking Report and a Usability Report. The Design Thinking Report will demonstrate the iterative design process, showing our work at each stage of development, and the reasoning behind each important decision we made. The Usability Report will take data collected from stakeholder testers to assess the usability of our system and determine any potential improvements that could be made.

2.4 Relevant Documentation

- Relevant documents include:

Problem Statement and Goals. This document includes information about our goals, which is what the Verification and Validation Plan's objectives are based on. It also lists stakeholders, which are relevant to our testing efforts.

Development Plan. This document describes our plan for our Proof of Concept Demonstration, which again highlights our most important objectives.

Software Requirements Specification. This document details all of the requirements, most of which will be subject to verification.

Hazard Analysis. This document includes some safety and security focused software requirements, which will be subject to verification.

Module Guide. This document describes all of the modules of our system. This is important for organizing our System and Unit Tests.

Module Interface Specification. This document describes specifications for the interfaces. This is also important for organizing our System and Unit Tests.

3 Plan

3.1 Verification and Validation Team

Table 1: Verification & Validation Team

Name	**Role**	**Task**	**Primary Section(s)**	**Secondary Section(s)**
Felix Hurst	Author	Team leader, focuses on keeping all data organized and spearheading meetings about testing with supervisor and other testers. Manages issue tracker. Secondary reviewer to all documents	N/A	All
BoWen Liu	Author	Primary Reviewer for engine and tool system. Secondary reviewer for art direction and environment direction.	Game Engine & Physics	Art & Environment
Andy Liang	Author	Primary Reviewer for Art, Environment, and Music systems. Secondary reviewer for all documents.	Art, Environment, Music	All
Marcos Hernandez-Rivero	Author	Primary reviewer for Unit testing and static analysis system, ensuring the systems are functional and up to date. Secondary reviewer for all docs.	Unit Testing, Static Analysis	All
Stephen Kelley	Supervisor	Capstone Supervisor, ensuring we are on track with the theme of the project, and serving as a reviewer for document completeness and correctness.	All	N/A
Michelle Bunton	Supervisor	Aid to the capstone supervisor, ensuring verification and validation of specific elements, such as the slime mold algorithm, are properly done.	All	N/A

3.2 SRS Verification

We intend to use a mix of ad hoc feedback from fellow team members and our supervisor, and a proper structured plan. This will be done via a structured inspection process involving both internal team reviews and external stakeholder feedback. Internally, team members will use an ad hoc feedback approach to log any issues concerning clarity or consistency in our

central issue tracker. For our supervisor, we will conduct a formal meeting where we present the game concept and critical requirements, followed by a Task–Based Inspection. The supervisor will be asked to specifically review pre–selected, complex requirements related to environment manipulation, physics simulations, and various elements of our project, such as the slime mold algorithm that we are using, focusing on their feasibility and testability. This ensures a structured approach to leveraging our supervisor's expertise beyond a simple read–over.

Table 2: SRS Verification Checklist

Criteria	**Description**
Completeness	Are all necessary functional and non-functional requirements (performance, usability, etc.) documented?
Correctness	Does each requirement accurately reflect the intended system and stakeholder needs?
Clarity/Ambiguity	Is each requirement stated unambiguously, with a single interpretation? (e.g., avoid words like "good," "usually," "robust")
Consistency	Are all requirements consistent with each other, and are terms/definitions used uniformly throughout?
Verifiability	Can a test procedure (inspection, test, demonstration) be designed to verify this requirement? (e.g., "The game must run at 60 FPS" is verifiable; "The game must be fast" is not).
Feasibility	Can the requirement be achieved with available technology, resources, and schedule?
Traceability	Is each requirement uniquely identified to allow tracing to design and test cases?

3.3 Design Verification

Our design verification plan focuses on ensuring the architectural and detailed design correctly implements the SRS and follows good software engineering principles like modularity. The core of this verification will be formal design inspections and walk–throughs. The Lead Developer will walk the team and classmates through critical design elements, such as the Tool–Effect System Architecture and the Level Data Structure, to scrutinize interface specifications and component interactions. Classmates will be leveraged as peer reviewers to ensure the design is robust and maintainable. The review process will focus on whether the design efficiently supports all functional requirements for traversing and manipulating the puzzle environments.

Table 3: Design Verification Checklist

Criteria	**Description**
Traceability to SRS	Does the design cover every requirement in the SRS, and is all design justified by a requirement?
Modularity & Coupling	Are components (classes, modules) logically grouped, and is the dependency/communication between them minimal and clearly defined?
Interface Specification	Are all internal and external component interfaces (APIs, data structures) fully and correctly defined?
Error Handling	Are potential error conditions (e.g., resource limit reached, invalid input) identified, and is a recovery/reporting mechanism defined in the design?
Performance/Resource	Does the design approach (e.g., algorithm choice, data structure) meet critical non-functional requirements like frame rate and memory usage?
Maintainability	Is the design simple, consistent, and easy to modify or extend without causing unintended side effects?

3.4 Verification and Validation Plan Verification

We will subject this document to a formal peer review by classmates, ensuring the plan adequately covers verification for all key project artifacts (SRS, Design, Implementation) and specifies a viable strategy for testing and validation. Additionally, we will employ conceptual mutation testing by deliberately considering how the plan would handle various types of software failures (e.g., unit test failure, non–functional requirement breach). We will also be subject to a review by Dr. Kelley and Michelle Bunton, to ensure that the core ideas and themes of the game, which may not be understood by classmates, are conveyed properly and free from errors in the document, as well as proper testing mechanisms for aspects which cannot be unit tested, like slime mold algorithm.

Table 4: Verification and Validation Plan Checklist

Criteria	**Description**
Scope Coverage	Does the V&V plan address verification for all major documents (such as SRS, design, code, etc) and all requirements (FR, NFR)?
Method Suitability	Are the chosen methods appropriate for the documents being verified?
Resource Allocation	Are the necessary tools, personnel (reviewers, testers), and time for each V&V activity specified and feasible?
SMA Testing	Does the plan specifically address the unique challenges of testing non-deterministic behavior like the SMA?
Acceptance Criteria	Does the plan clearly define the Pass/Fail criteria for system validation (e.g., what metrics define acceptable frame rate, or successful puzzle completion)?

3.5 Implementation Verification

This verification relies heavily on three components: automated testing, manual testing, and static analysis. We will execute all automated tests detailed in the Unit Testing Plan, covering critical logic like physics calculations and tool–effect application, that can be mathematically verified. For static verification, the team will conduct internal code inspections on complex components, such as the logic for object mutation. Furthermore, the final presentation in CAS 741 will be used as both a formal Code Walk–Through and to help with one of our extras which was a usability report, where a section of the most complex game logic is presented and explained to an external audience, and a usability survey can also be conducted. For manual tests (integration testing), we need to ensure that the game components connect well, and that the goals we set are being met such as framerate, graphics, tools working properly under different conditions, etc.

3.6 Automated Testing and Verification Tools

Our team will be using a combination of runtime and CI/CD tests to test each of our feature modules. For runtime tests, we will be using Unity's built-in Test Framework to create and run unit tests for our core game mechanics, such as proper geometry and clipping detection for terrain destruction system and slime mold traversal algorithm location and logic checking. These tests will be executed locally by developers during development to catch issues early. In terms of CI/CD tests, we will integrate automated test runs into our GitHub Actions pipeline for example formatting checks using tools like EditorConfig and static code analysis using SonarQube. This will ensure that every code change is automatically verified for both functionality and code quality before being merged into the main branch. Additionally, we will use Unity's Profiler to monitor performance metrics during gameplay, ensuring that non-functional requirements like frame rate and memory usage are consistently met.

3.7 Software Validation

We will use the Rev 0 demo to our supervisor/stakeholder as the first key opportunity for requirements validation, ensuring the core mechanics align with their expectations. If we are able to, we will also be demoing to a small group of external user testers, likely our friends or classmates in engineering programs, that can provide us with feedback. These users will be engaged in task–based inspection, where they are asked to complete specific puzzle levels while being observed. Feedback will be gathered on the usability of the tool–selection interface and the satisfaction of the core puzzle loop, directly validating the non–functional and game design requirements. We will then ask them questions based on their experience, and seeing what aspects they enjoyed or did not enjoy, and what they found easy or difficult to use.

4 System Tests

System tests provide an effective and quantifiable way of evaluating success of functional and non–functional requirements.

4.1 Tests for Functional Requirements

The following are tests for the functional requirements outlined in the SRS. Testing for functional requirements will include the two key game mechanics of destructive environment and path traversal as well as the saving feature. Each feature and mechanics' specific functionalities will be tested to ensure it meets proper adherence to functionality.

4.1.1 SavingManualTest-FRT01

Type: Functional, Dynamic, Manual

Initial State: Character at beginning of level.

Input: Move character to designated save point, destroy 2 objects. clicks save option from menu.

Output: Character coordinates, inventory, attributes, world state and game objects, and location saved to file.

Test Case Derivation: The saved information represents a snapshot of the player character to be loaded again to continue where the player has been left off.

How test will be performed: The user will move the character from point A to point B and perform the intended environmental manipulation and then click save. The user loads the game save and checks if the current state is maintained.

4.1.2 SavingAutoTest-FRT02

Type: Functional, Dynamic, Manual

Initial State: Character at beginning of level.

Input: Picks up 2 objects which will trigger a preprogrammed autosave save.

Output: Character coordinates, inventory, attributes, world state and game objects and location saved to file.

Test Case Derivation: The saved information represents a snapshot of the player character to be loaded again to continue where the player has been left off.

How the test will be performed: User will move the character from point A to point B and pick up the designated objects which will trigger save. The user loads the game save and checks if the current state is maintained.

4.1.3 DestructibleEnvironmentTest-FRT03

Type: Functional, Dynamic, Manual

Initial State: Character at beginning of level

Input: Go to designated point and use tool to destroy object at marked entry point. Repeat process with each tool and 3 different object attributes(soft object, brittle object, and hard object).

Output: Deformed original game object and spawned debris objects

Test Case Derivation: The deformed original game objects as well as the spawned game objects display shape deformation characteristics of the tool used and object attributes.

How the test will be performed: User will move the character to designated point and use tool to destroy object at marked entry point. Repeat process with each tool and 3 different object attributes(soft object, brittle object, and hard object).

4.1.4 SlimeMoldTraversalTest-FRT04

Type: Functional, Dynamic, Automatic

Initial State: Slime mold at spawned point, water debris at spawned point. 5 rectangular obstacles in the level each varying in size and dimensions, and one gap in terrain.

Input: Default slime mold entity spawn.

Output: Slime mold segments traverse to the water debris through the obstacles.

Test Case Derivation: Slime mold segments should travel from start to water debris objective in different and non deterministic ways each time such that the pathways traversed will be unique.

How test will be performed: Level will be set up with the game objects with the specified initial state and the slime mold will traverse to the water debris. Screenshot taken of each run of the experiment.

4.2 Tests for Nonfunctional Requirements

The following are tests for the nonfunctional requirements outlined in the SRS.

4.2.1 SaveGameIntegrityTest-NFRT01

Type: Non–Functional, Static, Automatic

Initial State: Game Saved to file.

Input/Condition: Run save file through decompression and deserialization check.

Output/Result: Console logged logical checks for each aspect of save file(player location, inventory, state, world object locations, etc)

How test will be performed: At a given point midway through a level, player will save the game, the save file will be run through our decompression and deserialization check and be evaluated if it passes all the logical checks.

4.2.2 DestructibleEnvironmentUsabilityTest-NFRT02

Type: Non–Functional, Dynamic, Manual

Initial State: Isolated Segments of levels, whole levels

Input: In the given test environment, the player will traverse from point A to B and is required to destroy certain terrain and objects between the traversal.

Output: Final game objects and terrain shape, generated debris, player traversal from point A to B

How test will be performed: In the given test environment setup to test critical sections of levels, the player will traverse from point A to B and is required to destroy certain terrain and objects between the traversal. Player actions are observed and are requested to fill out a survey (part of Section 6) with questions about traversability, issues with level design and destruction mechanics.

4.2.3 SlimeMoldAlgorithmBehaviorTest-NFRT03

Type: Non–Functional, Dynamic, Automatic

Initial State: Slime mold at spawned point, water debris at spawned point. Obstacles in level 5 varying in size and width boxes, and one gap in terrain.

Input: N/A

Output: Slime mold traversal from spawned point to desired end point, Screenshot of final result.

How test will be performed: 3 Trials will be conducted with the same test level. Observations on traversability, randomness, and evaluation comments will be made in a survey(part of section 6) by the developer.

4.3 Traceability Between Test Cases and Requirements

The following is a table for the test cases as they relate to the requirements.

5 Unit Test Description

5.1 Unit Testing Scope

Unit tests will focus on verifying individual modules and functions in isolation. The scope includes testing core game mechanics, physics calculations, save/load functionality, and algorithm implementations. Unit tests will be automated using NUnit or xUnit framework and will be executed as part of our continuous integration pipeline.

5.2 Tests for Functional Requirements

5.2.1 SaveLoadModule-UT01

Type: Functional, Dynamic, Automatic

Initial State: Empty save file system

Input: Player state object with coordinates (100, 200), inventory containing 3 items, and level ID `Level_2`

Output: Serialized save file containing all input data, successfully deserialized to match original state

Test Case Derivation: The save/load system must accurately preserve all player state data without loss or corruption. This test verifies serialization and deserialization correctness.

How test will be performed: Create a mock player state object, serialize it using the save module, deserialize it, and assert that all fields match the original values using NUnit assertions.

5.2.2 SaveLoadModule-UT02

Type: Functional, Dynamic, Automatic

Initial State: Existing save file with player at position (50, 75)

Input: Updated player state with new position (150, 225) and additional inventory item

Output: Updated save file that overwrites previous data correctly

Test Case Derivation: The save system must handle overwriting existing saves without data corruption or file system errors.

How test will be performed: Create initial save, modify player state, save again, then load and verify that only the new state is present and old data is properly overwritten.

5.2.3 PhysicsModule-UT01

Type: Functional, Dynamic, Automatic

Initial State: Debris object with mass 10kg at rest

Input: Apply force vector (100N, 0N) for 1 second

Output: Object velocity of (10 m/s, 0 m/s) based on F=ma

Test Case Derivation: Physics calculations must follow Newton's laws. With F=100N, m=10kg, t=1s, expected velocity is v = (F/m)*t = 10 m/s.

How test will be performed: Create mock debris object, apply force through physics module, measure resulting velocity, and assert it matches calculated expected value within 0.01 m/s tolerance.

5.2.4 PhysicsModule-UT02

Type: Functional, Dynamic, Automatic

Initial State: Two debris objects at positions (0, 0) and (5, 0) with collision detection enabled

Input: Move first object to position (5, 0)

Output: Collision event triggered, both objects have non-zero separation

Test Case Derivation: Collision detection must identify overlapping objects and trigger appropriate collision response.

How test will be performed: Position objects to overlap, call collision detection module, verify collision event is raised and objects are separated by physics resolution.

5.2.5 DestructionModule-UT01

Type: Functional, Dynamic, Automatic

Initial State: Intact terrain object of type SOFT at position (100, 100)

Input: Tool impact at position (100, 100) with force 50N and tool type HAMMER

Output: Terrain object deformed, 10-15 debris objects spawned within radius of 20 units

Test Case Derivation: Destruction system must generate debris based on material type and impact force. SOFT material with HAMMER should produce 10-15 chunks.

How test will be performed: Apply tool impact to terrain object, count generated debris objects, verify count is within expected range and debris positions are within impact radius.

5.2.6 DestructionModule-UT02

Type: Functional, Dynamic, Automatic

Initial State: Intact terrain object of type BRITTLE

Input: Tool impact with force 30N

Output: Complete object destruction, 20-30 small debris pieces spawned

Test Case Derivation: BRITTLE material should shatter more completely than SOFT material under similar force, producing more numerous smaller pieces.

How test will be performed: Apply impact to BRITTLE object, verify object is removed from scene, count debris (should be 20-30), and verify debris size is smaller than SOFT material debris.

5.2.7 SlimeMoldModule-UT01

Type: Functional, Dynamic, Automatic

Initial State: Single slime mold segment at position (0, 0), attractant at position (100, 0)

Input: Run algorithm for 10 iterations

Output: Slime mold segment has moved closer to attractant (x-coordinate increased)

Test Case Derivation: Slime mold algorithm should exhibit chemotaxis toward attractant, resulting in net movement in the correct direction over multiple iterations.

How test will be performed: Initialize slime mold and attractant, run algorithm iterations, verify final x-position is greater than initial position, indicating movement toward target.

5.2.8 SlimeMoldModule-UT02

Type: Functional, Dynamic, Automatic

Initial State: Slime mold at (0, 0), obstacle at (50, 0), attractant at (100, 0)

Input: Run algorithm for 50 iterations

Output: Slime mold reaches attractant without intersecting obstacle

Test Case Derivation: Algorithm must implement obstacle avoidance while maintaining progress toward target.

How test will be performed: Place obstacle between start and goal, run algorithm, verify slime mold path does not intersect obstacle collision bounds and eventually reaches target position.

5.3 Tests for Nonfunctional Requirements

5.3.1 PerformanceModule-UT01

Type: Non–Functional, Dynamic, Automatic

Initial State: Scene with 100 debris objects

Input/Condition: Execute one physics update cycle

Output/Result: Update completes in under 16ms (60 FPS target)

How test will be performed: Create scene with 100 debris objects, measure execution time of physics update using high-resolution timer, assert time is below 16ms threshold. Run 100 times and verify 95th percentile meets requirement.

5.3.2 PerformanceModule-UT02

Type: Non–Functional, Dynamic, Automatic

Initial State: Terrain with 50 destructible objects

Input/Condition: Destroy 10 objects simultaneously

Output/Result: Frame time remains under 33ms (30 FPS minimum), no frame drops

How test will be performed: Trigger destruction of 10 objects in single frame, monitor frame time during destruction and subsequent 30 frames, verify no frame exceeds 33ms.

5.3.3 MemoryModule-UT01

Type: Non–Functional, Dynamic, Automatic

Initial State: Game loaded with no debris

Input/Condition: Spawn 1000 debris objects then destroy all

Output/Result: Memory returns to within 10MB of initial state

How test will be performed: Measure baseline memory usage, spawn and destroy debris objects, force garbage collection, measure final memory usage, verify difference is within 10MB indicating no major memory leaks.

5.3.4 SaveLoadModule-UT03

Type: Non–Functional, Dynamic, Automatic

Initial State: Complex game state with 50 objects and full inventory

Input/Condition: Execute save operation

Output/Result: Save completes in under 500ms

How test will be performed: Create complex game state, measure time for complete save operation using stopwatch, assert time is under 500ms to ensure responsive save system.

5.4 Traceability Between Test Cases and Modules

Table 5: Unit Test to Module Traceability

Test ID	**Module**	**Purpose**
SaveLoadModule-UT01	Save/Load System	Verify serialization correctness
SaveLoadModule-UT02	Save/Load System	Verify overwrite functionality
SaveLoadModule-UT03	Save/Load System	Verify save performance
PhysicsModule-UT01	Physics Engine	Verify force application
PhysicsModule-UT02	Physics Engine	Verify collision detection
DestructionModule-UT01	Destruction System	Verify SOFT material behavior
DestructionModule-UT02	Destruction System	Verify BRITTLE material behavior
SlimeMoldModule-UT01	Slime Mold Algorithm	Verify basic chemotaxis
SlimeMoldModule-UT02	Slime Mold Algorithm	Verify obstacle avoidance
PerformanceModule-UT01	Physics Engine	Verify physics performance
PerformanceModule-UT02	Destruction System	Verify destruction performance
MemoryModule-UT01	Object Management	Verify memory management

5.5 Traceability Between Test Cases and Modules [Provide evidence that all of the modules have been considered. —SS]

References

Author Author. System requirements specification. https://github.com/..., 2019.

6 Appendix

This is where you can place additional information.

6.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC CONSTANTS. Their values are defined in this section for easy maintenance.

The following symbolic constants are referenced by system and non–functional tests (Sec. 4). Centralizing them here keeps test cases consistent if values change.

Table 6: Symbolic Parameters

Symbol	Definition	Value/Domain	Referenced Tests
SAVE_ON_DESTROY_COUNT	Objects destroyed (precondition) before manual save	2	FRTO1
AUTOSAVE_PICKUP_COUNT	Objects picked up to trigger auto-save	2	FRTO2
SAVE_RECORD_FIELDS	Fields persisted in a save snapshot	{PLAYER_COORDS, PLAYER_INVENTORY, PLAYER_ATTRIBUTES, WORLD_STATE, GAME_OBJECTS, LEVEL_LOCATION}	FRTO1, FRTO2
MATERIAL_TYPES	Object attribute classes for destruction tests	{SOFT, BRITTLE, HARD}	FRTO3
SLIME_MOLD_OBSTACLE_COUNT	Rectangular obstacles in path-traversal level	5	FRTO4
TERRAIN_GAP_COUNT	Terrain gaps in path-traversal level	1	FRTO4
SLIME_MOLD_ATTRACTANT	Target resource guiding traversal	WATER_DEBRIS	FRTO4

Notes: SAVE_RECORD_FIELDS mirrors the save–snapshot contents described in the save tests; "LEVEL_LOCATION" denotes the level/area identifier distinct from PLAYER_COORDS.

6.2 Usability Survey Questions

6.2.1 Operational Definitions

• Fairness (puzzles): The reason for failure/success is understandable; rules/hints are telegraphed; no pixel–perfect timing is required; a recovery path exists.

• Soft–lock: Progress cannot be made without a menu reset or hard restart. Recovering within $\leq$60 s using in–game options does not count as a soft–lock.

6.2.2 Purpose

This survey evaluates whether first-time average players can successfully learn controls, read the UI at the display baseline, understand organism responses to environmental stimuli (water/heat/light), complete the intended experience within target playtime, and encounter minimal stability/friction issues. It also collects qualitative feedback to guide UX fixes.

6.2.3 Participant Screener & Setup (Pre-Play)

Target Demographic: Gamers with baseline familiarity of 2D platformers/puzzle-platformers.

- Screener questions (check all that apply):

Q1. In the past 12 months, how many hours have you played 2D platformers/puzzle-platformers? 0, 1–9, 0–200, ¿200

Q2. Primary input used when playing today: Keyboard+Mouse Gamepad (XInput-style) Both

Q3. OS: Windows 10 (64-bit) Windows 11 Other/Unknown

Q4. Display resolution used: 1280×720 Higher Lower

Q5. Optional: Your PC specs if known (CPU / RAM / GPU)

Test setup notes. Provide players with either keyboard+mouse or gamepad input depending on their preference. Ensure display resolution is at least 1280×720.

6.2.4 Core Usability (5-point Likert; 1=Strongly disagree . . . 5=Strongly agree)

Q6. The basic controls were easy to learn.

Q7. The UI text/HUD were readable at my resolution (especially at 1280×720).

Q8. The organism reacted to water/heat/light in ways I could anticipate. The game responded quickly to my actions (no noticeable delay after stimuli).

Q9. I could play entirely with keyboard+mouse if needed (gamepad optional).

Q10.The overall first-time session length felt right for a 60–90 minute experience.

Q11. The game's message about collaborating with nature came through clearly.

6.2.5 Difficulty & Flow

Q12. The puzzles were challenging but fair (I knew why I failed/succeeded).

Q13. I rarely felt soft-locked or irrecoverably stuck.

Q14. The Reset Level option helped me recover when I got lost or stuck.

6.2.6 Stability & System UX

Q15. The game did not crash during my session.

Q16If a crash occurred, the crash dialog was informative and helpful.

Q17. The game ran without asking for administrator privileges.

6.2.7 Open-Ended Questions (Free Text)

Q18. What (if anything) was confusing about the organism's reactions to water/heat/light?

Q19. Where did you get stuck, and what would have helped?

Q20. Any places where text/UI felt too small or low-contrast? (Include your resolution.) What single change would most improve your first-time experience?

6.2.8 Observer Metrics (Facilitator fills; not asked to the player)

• Total session time (min); compare to target 60–90 min (P90 $\leq$ 120). • Crashes (#) and any crash dialog screenshots. • Resets used (#) and soft-lock observations. • Latency notes after stimuli (subjective: "felt instant/laggy"). • Environment: OS version, display resolution, input device(s), notable hardware.

6.2.9 Administration & Scale

• Use a 5-point Likert scale for Items 6–18. • Outline player background, choice of input device, and display resolution in screener. • Sample size for participants: minimum 10 for initial feedback; 20+ for statistical significance.

6.2.10 Traceability Examples (questions $\leftrightarrow$ requirement/goal)

• UI readability @1280×720 $\rightarrow$ Q7; • Input controls (Keyboard and mouse, and gamepad optional) $\rightarrow$ Q9. Maps to S6-UI-1. • Response Feedback $\rightarrow$ Q10; Maps to S6-IO-1 • Reset Puzzle to avoid soft-locks $\rightarrow$ Q14. Maps to S6-UNDO-1 • Stability $\rightarrow$ Q15–17. Maps to S6-PERF-1, S6-MEM-1, and S6-LOAD-1

7 Appendix — Reflection

1. What went well while writing this deliverable?

2. What pain points did you experience during this deliverable, and how did you resolve them?

3. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.

4. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?

Felix Hurst

1. It was easy to determine our project's goals, as our team had just recently met with our supervisors and are in agreement about the direction we want to take our development as well as our verification and validation processes.

2. There were some setbacks due to many of our team members devoting time to midterm examinations. In addition, some sections required previous sections to be filled out first before they could be finished. These issues were resolved through effective communication and time management.

3. I will need to learn about how to interpret the results from surveys, as they can be ambiguous and difficult to understand, due to their often qualitative nature rather than quantitative. This is important for gauging usability, which will be done through surveys.

- 4. Two approaches to acquiring this knowledge include:

– Running a practice survey among our team and attempting to interpret the results

– Reading texts online that describe how to interpret survey data

I will pursue the first approach, as I believe a first–hand experience would be more effective for my learning, and could also give us useful insights about our own project.

BoWen Liu

1. What went well while writing this deliverable?

Identifying the testing methodology and steps for each requirement went well due to our group having a solid grasp of how we want to implement the features.

2. What pain points did you experience during this deliverable, and howdid you resolve them?

A pain point of this deliverable was deciding how we want to evaluate success in each test and as such we decided to implement a checklist type system for functional requirements as there are solid requirements and a questionnaire type system for nonfunctional requirements as it cannot solely be evaluated well with a checklist to express the nuances.

3. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project?

Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc.

You should look to identify at least one item for each team member.

Knowledge of BtoC testing was important in regards to the nature of our product which in addition to needing to work well also contains many qualitative aspects which requires deeper nuance and breadth in testing.

4. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?

An approach to understanding and improving knowledge on BtoC testing is continuous feedback from peers and the market in terms of continuous product development. Another approach to acquiring said knowledge is to do plenty of research online in regards to similar type and scale of videogame companies to understand their approaches

Marcos Hernandez–Rivero

1. What went well while writing this deliverable?

We generally already knew exactly what we wanted to write and how we wanted to approach this document in terms of "Where are we going with this". On friday we had a meeting with our Capstone Supervisor so we were fresh from receiving feedback.

2. What pain points did you experience during this deliverable, and how did you resolve them?

For our type of project, we have a lot of aspects that are very hard to test. Stuff like the non–deterministic Slime Mold Algorithm, or having A LOT of non functional requirements means figuring out ways to test all of this is very hard. We resolved this by meeting as a team and brainstorming possible ways to solve this, but that could not get us all the way there for some things. Another pain point was that there were quite a few dependencies in this document, with some parts being uncompletable until other people completed a specific part. That combined with the previous week being super busy due to midterms, assignments,

etc means we were not able to dedicate as much time to this as we would have liked. This was mostly resolved by simply doing it on the weekend/on monday

3. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.

I will need to learn about dynamic testing, as many issues do not occur statically when each section is separate, but rather once the code is running and all the aspects of the projects are combined. I need to be able to determine how to troubleshoot, test, and fix issues when they appear in live runs.

4. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?

I see two main paths to obtain and practice this skill. The first one is simply watching videos and documentation on dynamic testing, especially for video games, which is likely to help somewhat but also very prone to being out of scope for our project depending on the contents of the document. The second approach I see is going through old projects (or prototypes for this project) and running dynamic testing by fault injection, where I manually add errors into the code so when the program runs something will be wrong, then I can determine if my dynamic testing skills can catch it. I feel like I will likely go with option 2, although having option 1 as a supplement is also beneficial.

Andy Liang

1. I set up a tight Req→Test→Evidence traceability view that exposed gaps early and kept Section 6 internally consistent. I also translated subjective non–functional requirements into measurable thresholds (e.g., viewing distance, contrast ratio, legibility size) and centralized symbolic parameters (Sec. 6.1) so all tests referenced the same constants.

2. Defining objective oracles for behaviors that shouldn't assert exact paths and for UX that felt subjective was difficult. I resolved this by using seedable, property–based checks with acceptance envelopes (reach target within N steps, avoid forbidden cells, show monotonic progress) and by writing operational UX definitions (distance, contrast, error rate) that became pass/fail criteria in our checklists and questionnaires, with snapshot logs for run–to–run diffs.

3. I will need to learn writing testable requirements and accessibility verification (contrast/legibility, cue redundancy).

4. I will (a) run a "requirement clinic" pass to make statements measurable and (b) build an accessibility checklist—I will do (a) first to lock clarity, then (b) to keep UX tests objective.