Library of Lighting Models: System Verification and Validation Plan for Family of Lighting Models

Sasha Soraine

November 4, 2019

1 Revision History

Date	Version	Notes
October 17, 2019	1.0	Original Draft.

Contents

List of Tables

List of Figures

2 Symbols, Abbreviations and Acronyms

symbol	**description**
T	Test

[symbols, abbreviations or acronyms – you can simply reference the SRS tables, if appropriate —SS]

This document outlines a system validation and verification plan for the implementation of a sub-family of lighting models, based on the Commonality Analysis for a Family of Lighting Models. First it will cover general information about the system, including the particular design qualities the system should emphasize and any relevant documentation. Next it will outline the verification plans for the commonality analysis/requirements, system design, and implementation. It will then outline the software validation plan. Finally it will outline a series of representative test cases that are meant to test the functional and non-functional requirements, along with a traceability matrix mapping the test cases to particular requirements.

3 General Information

3.1 Summary

This software implements a sub-family of lighting models. The larger family and problem analysis is found in **??**. This software aims to take in user specifications about the graphical scene (lights, objects, shading model, lighting model, and an observer) and render a fully lit and shaded scene. To do this is runs calculations using basic optics principles to approximate light behaviour in 3D computer graphics.

3.2 Objectives

The goal of this testing is to:

- Demonstrate adequate usability of the system,
- Demonstrate learnability of the system, and
- Evaluate the productivity of the system.

3.3 Relevant Documentation

This section outlines other documentation that is relevant for understanding this document.

Document Name	Document Type	Document Purpose	Citation
Commonality Analysis of a Family of Lighting Models	Commonality Analysis	Problem domain description, and scoping to a reasonable implementation size through assumptions and requirements.	

4 Plan

This section outlines the verification and validation plans, including any techniques or data sets being used in the testing process. It also outlines the members of the verification and validations team.

4.1 Verification and Validation Team

This section lists the members of the verification and validation team. These are individuals who contribute to the verification and validation of the system and software design. Individuals listed here have specific roles denoting the amount of involvement they will be having in the verification and validation process. Primary roles are actively working on it; secondary roles view the system when major submissions are made; tertiary roles are asked to contribute if able, but are under no obligation to participate.

The verification and validation team includes:

Name	Role	Goal
Sasha Soraine	Primary Reviewer	Ensure the verification and validation process runs smoothly.
Peter Michalski	Secondary Reviewer	Ensure the logical consistency of system design and requirements in accordance with feedback role as expert reviewer.
Dr. Spencer Smith	Secondary Reviewer	Ensure reasonable coverage of design considerations and requirements as part of marking these documents.
CAS 741 Students	Tertiary Reviewers	Ensure general consistency in design and requirements coverage in accordance with feedback role as secondary reviewers.

4.2 CA Verification Plan

We aim to verify the requirements listed in the Commonality Analysis in the following ways:

- Have expert level users (familiar with graphics programming) do a close read of the commonality analysis to compare it against existing software tools for functionality.
- Review and revise requirements based on feedback from Domain Expert and Secondary Reviewer of CA.
- Ask Dr. Smith to review the scope to consider whether the implementation scoping and thus listed requirements is inappropriate.

4.3 Design Verification Plan

We will be using the following methods to test the design:

- Rubber duck testing,
- Expert review,
-

4.4 Implementation Verification Plan

The purpose of the implementation verification plan is to perform functional testing of the components of the system. As such it measures whether the system is behaving inline with the requirements documentation, and whether it meets its non-functional requirements thresholds.

We will be using the following methods to test the functional implementation:

- Boundary value testing,
- Endurance testing,
- Error handling testing.

We will be using the following methods to test the non-functional implementation:

- Installation testing,
-

These are carried out through the test cases listed in this document.

4.5 Software Validation Plan

[If there is any external data that can be used for validation, you should point to it here. If there are no plans for validation, you should state that here. —SS]

5 System Test Description

5.1 Tests for Functional Requirements

The following sections outline test cases for the functional requirements of the system.

This section is divided into different testing areas.

[Subsets of the tests may be in related, so this section is divided into different areas. If there are no identifiable subsets for the tests, this level of document structure can be removed. —SS]

[Include a blurb here to explain why the subsections below cover the requirements. References to the SRS would be good. —SS]

5.1.1 Input Testing

[It would be nice to have a blurb here to explain why the subsections below cover the requirements. References to the SRS would be good. If a section covers tests for input constraints, you should reference the data constraints table in the SRS. —SS]

Load a Scene

The following test cases all share the same properties:

Initial State: No scene loaded.

1. loadScene-allValid

   Control: Automatic

   Input: *SCENE_DIR/valid-AllInputs.JSON*

   Output: *RENDERS_DIR/render-valid-AllInputs*

   Test Case Derivation: Having received a properly formatted and structured JSON file containing all inputs for the scene, the system will output a fully rendered and lit scene.

   How test will be performed: System will try to load created test file from scene directory.

2. loadScene-validMissingSome

   Control: Automatic

   Input: *SCENE_DIR/valid-MissingData.JSON*

   Output: Prompt Message: "File exists, but is missing data. Would you like to load the default settings for missing data?", Log message "Error: File exists but is missing data."

   Test Case Derivation:

   How test will be performed: System will try to load created test file from scene directory.

3. loadScene-fileExistNoData

   Control: Automatic

   Input: *SCENE_DIR/valid-NoData.JSON*

   Output: Prompt Message "File exists, but is empty. Would you like to load the default scene?", Log message "Error: File exists but is empty."

   Test Case Derivation: In receiving an empty file, the system should be robust enough to acknowledge the specific error and offer to substitute with the default scene.

   How test will be performed: System will try to load created test file from scene directory.

4. loadScene-invalidInput

   Control: Automatic

   Input: *SCENE_DIR/invalid.JSON*

   Output: Error Message"File does not contain valid scene data.", Log message "Error: Invalid scene data in file: *SCENE_DIR/invalid.JSON*."

   Test Case Derivation: In receiving invalid data, the system should be robust enough to acknowledge the specific error and log it in the log file.

   How test will be performed: System will try to load created test file from scene directory.

Create Default Scene

1. createDefault-validMissingData

   Control: Automatic

   Initial State: No scene loaded. System tried to load *SCENE_DIR/valid-MissingData.JSON* and displayed Prompt Message.

   Input: YES selected at Prompt Message

   Output: *RENDERS_DIR/render-valid-MissingData-defaultsub*

Test Case Derivation: System corrects partially invalid input by substituting missing information with predetermined default values.

How test will be performed: System will fill in empty input with default values.

2. createDefault-fileExistsNoData

   Control: Automatic

   Initial State: No scene loaded. System tried to load *SCENE_DIR/valid-NoData.JSON* and displayed associated Prompt Message.

   Input: YES selected at Prompt Message

   Output: *RENDERS_DIR/render-valid-NoData-defaultsub*

   Test Case Derivation: System corrects partially invalid input by substituting missing information with predetermined default values.

   How test will be performed: System will fill in empty input with default values.

5.1.2 Run-Time Tests

This subset of tests outline scenarios that may happen during the run-time of the program. As such it handles changes to the scene and rendering information.

All of these test cases share the following properties:

Initial State: A valid scene (*SCENE_DIR/valid-AllInputs.JSON*) is loaded to the system.

Lighting Model Changes

1. lightModel-valid

   Control: Automatic

   Input: Select different lighting model from dropdown list.

   Output: *RENDERS_DIR/render-valid-AllInputs* with visuals based on new lighting models.

Test Case Derivation: After a scene is loaded, the user can only interact with the system through its GUI. Lighting model changes are determined by dropdown list selection.

How test will be performed: System will recalculate luminous intensity of points on objects in the scene using the new model. New luminous intensity information will be sent through the rendering pipeline to output visuals.

Shading Model Changes

1. shadingModel-valid

   Control: Automatic

   Input: Select different shading model from dropdown list.

   Output: *RENDERS_DIR/render-valid-AllInputs* with visuals based on new shading models.

   Test Case Derivation: After a scene is loaded, the user can only interact with the system through its GUI. Shading model changes are determined by dropdown list selection.

   How test will be performed: System will recalculate surface normals of points on objects in the scene using the new model. New surface normal information will be sent through the rendering pipeline to output visuals.

Object Changes

1. objMaterialPropChange-valid-ks

   Control: Automatic

   Input: $k_s = 0.5$

   Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed so that specular reflection uses new value.

   Test Case Derivation: $I_s = k_s \cdot i(p, p_0) \cdot \max(0, (L_r \bullet V))^\alpha$. Final colouring of any point in a scene is $(I_a + I_d + I_s) \cdot LIGHT\_COLOUR$, therefore changes to the k_s impact the specular component of the final scene.

How test will be performed: Valid scene is loaded. Testing framework automatically assigns k_s the new value.

2. objMaterialPropChange-valid-kd

   Control: Automatic

   Input: $k_d = 0.5$

   Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed so that specular reflection uses new value.

   Test Case Derivation: $I_d = k_d \cdot i(p, p_0) \cdot \max(0, (L_i \bullet N))$. Final colouring of any point in a scene is $(I_a + I_d + I_s) \cdot LIGHT\_COLOUR$, therefore changes to the k_d impact the diffuse component of the final scene.

   How test will be performed: Valid scene is loaded. Testing framework automatically assigns k_d the new value.

3. objMaterialPropChange-valid-ka

   Control: Automatic

   Input: $k_a = 0.5$

   Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed so that specular reflection uses new value.

   Test Case Derivation: $I_a = k_a \cdot i(p, p_0)$. Final colouring of any point in a scene is $(I_a + I_d + I_s) \cdot LIGHT\_COLOUR$, therefore changes to the k_a impact the ambient component of the final scene.

   How test will be performed: Valid scene is loaded. Testing framework automatically assigns k_a the new value.

4. objMaterialPropChange-valid-α

   Control: Automatic

   Input: $\alpha = 2$

Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed so that specular reflection uses new value.

Test Case Derivation: $I_s = k_s \cdot i(p, p_0) \cdot \max(0, (L_r \bullet V))^{\alpha}$. Final colouring of any point in a scene is $(I_a + I_d + I_s) \cdot LIGHT\_COLOUR$, therefore changes to the α impact the specular component of the final scene.

How test will be performed: Valid scene is loaded. Testing framework automatically assigns α the new value.

5. objMaterialPropChange-invalid-ks

   Control: Automatic

   Input: $k_s = 2$

   Output: Error Message "New value of k_s is outside of bounds. Please enter a different value.". Log message "Error: tried to assign $k_s = 2$".

   Test Case Derivation: $0 \leq k_s \leq 1$; therefore the new assignment is invalid by the constraints.

   How test will be performed: Valid scene is loaded. Testing framework attempts to assign k_s the new value. Error message is loaded instead. Error is written to log file.

6. objMaterialPropChange-invalid-kd

   Control: Automatic

   Input: $k_d = 2$

   Output: Error Message "New value of k_d is outside of bounds. Please enter a different value.". Log message "Error: tried to assign $k_d = 2$".

   Test Case Derivation: $0 \leq k_d \leq 1$; therefore the new assignment is invalid by the constraints.

   How test will be performed: Valid scene is loaded. Testing framework attempts to assign k_d the new value. Error message is loaded instead. Error is written to log file.

7. objMaterialPropChange-invalid-ka

Control: Automatic

Input: $k_a = 2$

Output: Error Message "New value of k_a is outside of bounds. Please enter a different value.". Log message "Error: tried to assign $k_a = 2$".

Test Case Derivation: $0 \leq k_a \leq 1$; therefore the new assignment is invalid by the constraints.

How test will be performed: Valid scene is loaded. Testing framework attempts to assign k_a the new value. Error message is loaded instead. Error is written to log file.

8. objMaterialPropChange-invalid-α

   Control: Automatic

   Input: $\alpha = -2$

   Output: Error Message "New value of α is outside of bounds. Please enter a different value.". Log message "Error: tried to assign $\alpha = -2$".

   Test Case Derivation: $\alpha : \mathbb{Z}_+$; therefore the new assignment is invalid by the constraints.

   How test will be performed: Valid scene is loaded. Testing framework attempts to assign α the new value. Error message is loaded instead. Error is written to log file.

9. objMaterialPropChange-bound-ks

   Control: Automatic

   Input: $k_s = 1$

   Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed so that specular reflection uses new value.

   Test Case Derivation: $I_s = k_s \cdot i(p, p_0) \cdot \max(0, (L_r \bullet V))^\alpha$. Final colouring of any point in a scene is $(I_a + I_d + I_s) \cdot LIGHT\_COLOUR$, therefore changes to the k_s impact the specular component of the final scene.

   How test will be performed: Valid scene is loaded. Testing framework automatically assigns k_s the new value.

10. objMaterialPropChange-bound-kd

    Control: Automatic

    Input: $k_d = 1$

    Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed so that specular reflection uses new value.

    Test Case Derivation: $I_d = k_d \cdot i(p, p_0) \cdot \max(0, (L_i \bullet N))$. Final colouring of any point in a scene is $(I_a + I_d + I_s) \cdot LIGHT\_COLOUR$, therefore changes to the k_d impact the diffuse component of the final scene.

    How test will be performed: Valid scene is loaded. Testing framework automatically assigns k_d the new value.

11. objMaterialPropChange-bound-ka

    Control: Automatic

    Input: $k_a = 1$

    Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed so that specular reflection uses new value.

    Test Case Derivation: $I_a = k_a \cdot i(p, p_0)$. Final colouring of any point in a scene is $(I_a + I_d + I_s) \cdot LIGHT\_COLOUR$, therefore changes to the k_a impact the ambient component of the final scene.

    How test will be performed: Valid scene is loaded. Testing framework automatically assigns k_a the new value.

12. objMaterialPropChange-bound-α

    Control: Automatic

    Input: $\alpha = 0$

    Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed so that specular reflection uses new value.

    Test Case Derivation: $I_s = k_s \cdot i(p, p_0) \cdot \max(0, (L_r \bullet V))^{\alpha}$. Final colouring of any point in a scene is $(I_a + I_d + I_s) \cdot LIGHT\_COLOUR$, therefore changes to the α impact the specular component of the final scene.

How test will be performed: Valid scene is loaded. Testing framework automatically assigns α the new value. How test will be performed:

13. test-id2
    Change Object Position -valid

    Control: Manual versus Automatic

    Input:

    Output: [The expected result for the given inputs —SS]

    Test Case Derivation: [Justify the expected value given in the Output field —SS] How test will be performed:

14. test-id2
    Change Object Position - invalid

    Control: Manual versus Automatic

    Input:

    Output: [The expected result for the given inputs —SS]

    Test Case Derivation: [Justify the expected value given in the Output field —SS]

    How test will be performed:

15. test-id2
    Change Object Position -boundary

    Control: Manual versus Automatic

    Input:

    Output: [The expected result for the given inputs —SS]

    Test Case Derivation: [Justify the expected value given in the Output field —SS]

    How test will be performed:

16. test-id2
    Change Object Colour -valid

    Control: Manual versus Automatic

    Input:

Output: [The expected result for the given inputs —SS]

Test Case Derivation: [Justify the expected value given in the Output field —SS]

How test will be performed:

17. test-id2
    Change Object Colour - invalid

    Control: Manual versus Automatic

    Input:

    Output: [The expected result for the given inputs —SS]

    Test Case Derivation: [Justify the expected value given in the Output field —SS]

    How test will be performed:

18. test-id2
    Change Object Shape -valid

    Control: Manual versus Automatic

    Input:

    Output: [The expected result for the given inputs —SS]

    Test Case Derivation: [Justify the expected value given in the Output field —SS]

    How test will be performed:

19. test-id2
    Change Object Shape - invalid

    Control: Manual versus Automatic

    Input:

    Output: [The expected result for the given inputs —SS]

    Test Case Derivation: [Justify the expected value given in the Output field —SS]

    How test will be performed:

Light Changes

1. test-id2
   Change Light Position -valid

   Control: Manual versus Automatic

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2
   Change Light Position - invalid-out of room

   Control: Manual versus Automatic

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

3. test-id2
   Change Light Position -boundary

   Control: Manual versus Automatic

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

4. test-id2
   Change Light Position - valid-beside object

   Control: Manual versus Automatic

   Input:

Output: [The expected result for the given inputs —SS]

Test Case Derivation: [Justify the expected value given in the Output field —SS]

How test will be performed:

5. test-id2
   Change Light Position - valid behind object (obj between viewer and light)

   Control: Manual versus Automatic

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

6. test-id2
   Change Light Position - invalid-on top of viwere/object

   Control: Manual versus Automatic

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

7. test-id2
   Change Light Colour -valid

   Control: Manual versus Automatic

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

8. test-id2
   Change Light Colour - invalid

   Control: Manual versus Automatic

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

9. test-id2
   Change Light Shape -valid

   Control: Manual versus Automatic

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

10. test-id2
    Change Light Shape -valid-facing away from objects

    Control: Manual versus Automatic

    Input:

    Output: [The expected result for the given inputs —SS]

    Test Case Derivation: [Justify the expected value given in the Output field —SS]

    How test will be performed:

11. test-id2
    Change Light Shape - invalid

    Control: Manual versus Automatic

    Input:

    Output: [The expected result for the given inputs —SS]

Test Case Derivation: [Justify the expected value given in the Output field —SS]

How test will be performed:

5.2 Tests for Nonfunctional Requirements

[The nonfunctional requirements for accuracy will likely just reference the appropriate functional tests from above. The test cases should mention reporting the relative error for these tests. —SS]

[Tests related to usability could include conducting a usability test and survey. —SS]

5.2.1 Area of Testing1

Title for Test

1. test-id1

   Type:

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

5.2.2 Area of Testing2

...

5.3 Traceability Between Test Cases and Requirements

[Provide a table that shows which test cases are supporting which requirements. —SS]

References

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.

6.2 Usability Survey Questions?

[This is a section that would be appropriate for some projects. —SS]