Lights, Camera, Models!: System Verification and Validation Plan for Family of Lighting Models

Sasha Soraine

December 18, 2019

1 Revision History

Date	Version	Notes
October 17, 2019	0.1	Original Draft.
November 4, 2019	1.0	Version 1 - Submitted for class.
December 18, 2019	2.0	Version 2

Contents

List of Tables

List of Figures

2 Symbols, Abbreviations and Acronyms

2.1 Table of Units

Throughout this document SI (Système International d'Unités) is employed as the unit system. In addition to the basic units, several derived units are used as described below. For each unit, the symbol is given followed by a description of the unit and the SI name.

symbol	**unit**	**SI**
rad	angle	radian
cd	luminous intensity	candela

2.2 Table of Symbols

The table that follows summarizes the symbols used in this document along with their units. The choice of symbols was made to be consistent with the physics and calculus notation. The symbols are listed in alphabetical order.

symbol	**unit**	**description**
$\mathbb{R}$	—	Set of Real numbers.
$\mathbb{Z}$	—	Set of Integers.
$\mathbb{Z}_+$	—	Set of Positive Integers.
θ	rad	Angle between the viewer and the reflected ray.
p	—	Point in space or on a surface.
p_0	—	Point of origin.
v	—	Position of viewer represented as a 3D point in space.
L_i	—	Vector form of incident ray.
L_r	—	Vector form of reflected ray.
V	—	Vector from point to the viewer.
n	—	Vector form of surface normal.
N	—	Unit vector of surface normal, n.
H	—	Unit vector halfway between the incident ray, L_i, and the viewer vector, V.

I_a	cd	Ambient luminous intensity.
k_a	–	Coefficient of Ambient Reflection.
I_d	cd	Diffuse luminous intensity.
k_d	–	Coefficient of Diffuse Reflection.
I_s	cd	Specular luminous intensity.
k_s	–	Coefficient of Specular Reflection.
α	–	Shininess Coefficient.
I_T	cd	Total luminous intensity.

2.3 Abbreviations and Acronyms

symbol	**description**
GS	Goal Statement
LC	Likely Change
R	Requirement
CA	Commonality Analysis
LM	Lighting Model
Lights, Camera, Models!	Lights, Camera, Models!
T	Theoretical Model
3D	Three Dimensional
API	Application Programming Interface

This document outlines a system validation and verification plan for Lights, Camera, Models!, an implementation of a family of lighting models, based on the Commonality Analysis (CA) for a family of lighting models ?? `https://github.com/sorainsm/library-of-lighting-models/tree/master/Commonality-Analysis`. [I suggest a citation to your commonality analysis, and/or a link to the github location for the document. —SS] 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 CA. 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, it 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**
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:

[If you use the booktabs package, your tables will look a little better spaced out. (I also shortened the width of the last column of your table.) —SS]

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. [What existing tools are you comparing to? This would be very helpful information. —SS]
- Review and revise requirements based on feedback from Domain Expert and Secondary Reviewer of CA.
- Ask Dr. Smith [Dr. Smith includes two spaces after the period. Use Dr. Smith, or Dr. Smith —SS] to review the scope to consider whether the implementation scoping and thus listed requirements is inappropriate.

4.3 Design Verification Plan

The purpose of design verification is to ensure the structure of the code and design of the system meets the requirements laid out in CA.

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

- Rubber duck testing,

- Expert review,
- Task-based peer reviews.

The rubber duck testing will be performed by the primary tester (me). The rationale is that it should make holes in the design decisions apparent by forcing the primary tester to focus on justifying the system. The procedure will involve close examination of the code, with a spoken aloud explanation of the design decisions.

The expert review will be performed by the secondary reviewers (Peter and Dr. Smith). The rationale is that testers familiar with the project should be able to verify if the design meets the requirements and constraints. This will be done through posting of GitHub issues and feedback in the document.

The task-based peer reviews will be performed by the tertiary testers (CAS 741 peers). The rationale is that targeted examination of parts of the system are easier to perform and will generate better feedback. To divide the tasks, every aspect of the system has to fulfill some requirement - the testers will then examine whether that component of the system meets that requirement. The feedback will be captured via GitHub issues.

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:

- Rubber duck testing,
- Peer reviews,
- Expert reviews,
- Boundary value testing,
- Endurance testing,
- Error handling testing.

Rubber duck testing, peer reviews, and expert reviews will also be used as implementation verification techniques. These will be executed simultaneously with the design verification versions, as the close attention paid in that setting lends itself to looking for potential implementation errors.

Boundary value testing will outline the expected behaviour of the system at the boundaries of variable constraints. This testing ensures that all edge-case behaviour has been considered and that system responses are designed for those cases.

Error handling testing will outline the expected system behaviour when invalid data enters the system. This testing ensures that the system recognizes valid data from invalid data and provides appropriate feedback to the user. Successful error handling testing will also lead to better usability of the system as more feedback on how to correct the system is provided to users.

Endurance testing will outline the system behaviour under repetitive tasks with valid input. The purpose of this is to push the system to its load capacity and see how it reacts when its output keeps changing. The rationale for this testing is to test the reliability of the software.

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

- Installation testing,
- Usability testing.

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

These system tests are supplemented by the unit tests listed in CA. This system testing presupposes that the system has passed its unit and integration tests, and that the functionality of each part has been verified. This assumption allows us to interpret the results of these system tests as evaluating the design of the system structure as the sum of its parts.

[The implementation verification plan is very ambitious. You might have to “fake it” and pretend that some of the verification will be taking place after the course is over. —SS]

4.5 Software Validation Plan

There are currently no plans to validate the software. [Why not? I think you should explain that validation is not suitable for your software. You are not comparing the output to reality; you are comparing your results to mathematical models. —SS]

5 System Test Description

The following section outlines the test cases to be used for testing the functional and non-functional requirements at a system level.

These test cases are divided between the functional and non-functional requirements. The functional requirement test cases handle black-box behvaiour [spell check —SS] of the system as it is fed different inputs. The non-functional requirement test cases focus on testing the usability of the system as an end-user. Therefore it doesn't test the productivity and maintainability based non-functional requirements.

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. These are:

- Input testing, and
- Run-Time testing.

The subdivision was made to capture the types of tasks the system would need to anticipate at different points in time during use. Input testing is preliminary before the system begins to run calculations. Run-time testing is the real-time calculations that the system needs to make in response to user input.

There is no explicit output testing. By nature of this system rendering images, every functional test implicitly tests the ability to output a file. As such, the basic output testing is handled alongside the "Load a Scene" tasks.

5.1.1 Input Testing

The following section explicitly covers R1, R2, R3, R4, R5, and R6 [The original blank project template is set up to allow external cross-references between documents. Hard-coding these numbers will be a maintenance challenge. —SS] from the CA **??**. [It is a good idea to reference the CA, but the reference is missing. —SS] These requirements pertain to input handling; how the system should behave when various types of input are given to it. Implicitly these test R7, R8, R9, and R11 which are the underlying calculations that transform the input into the output scene.

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* (See 6.1)

   Output: *RENDERS_DIR/render-valid-AllInputs* [You should explain somewhere the SCENE_DIR and the RENDERS_DIR folders. Are these actual names, or are they placeholders that the tester will fill in with their specific folders? I prefer the second option, since it is more general. —SS]

   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. [How do you automatically assess success on this test? Do you simply look for the presence of a file in the expected location? A similar question applies for your other input tests. —SS]

2. loadScene-validMissingSome

   Control: Automatic

   Input: *SCENE_DIR/valid-MissingData.JSON*(See 6.1)

   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* (See 6.1)

   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* (See 6.1)

   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. [For the tests where an exception is expected, you can automate this in your unit testing by having the test case assertion be that the correct exception is raised. —SS]

Create Default Scene

1. createDefault-validMissingData

Control: Automatic

Initial State: No scene loaded. System tried to load *SCENE_DIR/valid-MissingData.JSON* (See 6.1) 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* (See 6.1) 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.

[I like how you have specific tests cases. Great! —SS]

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 equations pertaining to how that information is calculated can be found in the CA.

All of these test cases share the following properties:

Initial State: A valid scene (*SCENE_DIR/valid-AllInputs.JSON*)(See 6.1) is loaded to the system.

Lighting Model Changes

1. lightModel-valid

   Control: Automatic

   Input: Select different lighting model from dropdown list. [There isn't enough information here for someone else to do this test. The input is ambiguous. The same comment applies for other tests in this section. —SS]

   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. [How do you tell if this test has passed? How do you tell this automatically? —SS]

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. [How do you automatically tell that this test case passes? Are you just checking a single value, or how the value is used in the scene? A similar comment applies to other test cases in this section. —SS]

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: $0 \le k_s \le 1$; 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.

13. objPosition-valid

    Control: Automatic

    Input: New Point (2,0,0) for centre of object

    Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed to reflect object movement to coordinate (2,0,0) (including update of lighting as distance from light source is changed).

    Test Case Derivation: Lighting is dependent of position of object relative to the light source; therefore movement in position changes the lighting of an object causing all the intensities to be recalculated and the object to be recoloured.

    How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (2,0,0). System recalculates intensities and recolours moved object.

14. objPosition-invalid-outBounds

    Control: Automatic

    Input: New Point (11,0,0) for centre of object

    Output: Error Message "The centre of this object is outside of the room. It cannot be rendered. Please enter a different location for this object.". Log message "Error: Tried to move centre of object to (11,0,0).

Test Case Derivation: Scene size is defined to be (SCENE_HEIGHT, SCENE_WIDTH, SCENE_DEPTH); if any component of the object's position is greater than any component of the scene size then the object is out of bounds. Out of bounds objects cannot be lit or rendered.

How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (11,0,0). System throws an error.

15. objPosition-invalid-onLight

    Control: Automatic

    Input: New Point (5,5,5) for centre of object

    Output: Error Message "The centre of this object intersects the light source. It cannot be rendered. Please enter a different location for this object.". Log message "Error: Tried to move centre of object to light source position, (5,5,5)".

    Test Case Derivation: As per the requirement (**??** objects cannot be on top of the light source. This would create a design issue of whether the opaque object would be blocking the light. To circumvent this, we impose that they cannot have their centres at the same position.

    How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (5,5,5). System throws an error.

16. objPosition-invalid-onObserver

    Control: Automatic

    Input: New Point (0,0,0) for centre of object

    Output: Error Message "The centre of this object intersects the observer. It cannot be rendered. Please enter a different location for this object.". Log message "Error: Tried to move centre of object to observer position, (0,0,0)".

    Test Case Derivation: As per the requirement (**??** objects cannot be on top of the observer. This would create a design issue of whether the opaque object would be blocking the view, and how it would be lit. To circumvent this, we impose that they cannot have their centres at the same position.

How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (0,0,0). System throws an error.

17. objPosition-valid-edgeOfRoom

    Control: Automatic

    Input: New Point (0,10,0) for centre of object

    Output: Error Message "Parts of this object are outside of the scene size. Please move this object and re-render the scene.". Log Message "Error: Object partially outside of room."

    Test Case Derivation: Constraints on object position said $0 \ll x \leq SIZE\_HEIGHT$, which means that the centre is on the wall of the scene. This means part of the object would be rendered outside the scene.

    How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (0,10,0). System throws an error.

18. objPosition-valid-betweenLightAndViewer

    Control: Automatic

    Input: New Point (2,2,2) for centre of object

    Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed to reflect object movement to coordinate (2,2,2) (including update of lighting as distance from light source is changed).

    Test Case Derivation: Lighting is dependent of position of object relative to the light source; therefore movement in position changes the lighting of an object causing all the intensities to be recalculated and the object to be recoloured. When an object is between the light and the viewer its faces should be dark as they're not being lit.

    How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (2,2,2). System recalculates intensities and recolours moved object.

19. objPosition-valid-besideLight

    Control: Automatic

    Input: New Point (5,3,0) for centre of object

    Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed to reflect object movement to coordinate (5,3,0) (including update of lighting as distance from light source is changed).

    Test Case Derivation: Lighting is dependent of position of object relative to the light source; therefore movement in position changes the lighting of an object causing all the intensities to be recalculated and the object to be recoloured. When an object is beside a light source it is highly illuminated making it very bright.

    How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (5,3,0). System recalculates intensities and recolours moved object.

20. objColour-valid-base

    Control: Automatic

    Input: New (r,g,b) value for BASE_COLOUR picked from GUI picker = (10,255,50).

    Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed to reflect the new base colour for the object. All diffuse terms need to be recalculated.

    Test Case Derivation: The intensity of the BASE_COLOUR at a point is part of the diffuse model calculation. Change the BASE_COLOUR requires recalculating all of the intensity values with this new (r,g,b) information.

    How test will be performed: Valid scene is loaded. Testing framework automatically assigns new r,g,b to (10,255,50). System recalculates intensities and recolours object.

21. objColour-valid-specular

    Control: Automatic

Input: New (r,g,b) value for SPECULAR_COLOUR picked from GUI picker = (10,255,50).

Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed to reflect the new base colour for the object. All diffuse terms need to be recalculated.

Test Case Derivation: The intensity of the SPECULAR_COLOUR at a point is part of the specular component calculation. Change the SPECULAR_COLOUR requires recalculating all of the intensity values with this new (r,g,b) information.

How test will be performed: Valid scene is loaded. Testing framework automatically assigns new r,g,b to (10,255,50). System recalculates intensities and recolours object.

22. objShape-valid

    Control: Automatic

    Input: New shape selection from dropdown list: torus.

    Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed to replace the sphere with a torus. All normals have changed, so all intensities need to be recalculated.

    Test Case Derivation: Different shapes have different normals, and so different ways of reflecting. The models need to be sure to work with not just a sphere, but other objects as well.

    How test will be performed: Valid scene is loaded. Testing framework automatically assigns new object shape. System recalculates intensities and recolours new object.

Light Changes

1. lightPos-valid

    Control: Automatic

    Input: New Point (2,0,0) for centre of light source

Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed to reflect light source movement to coordinate (2,0,0) (including update of lighting as distance from light source is changed).

Test Case Derivation: Lighting is dependent of position of object relative to the light source; therefore movement in light source position changes the lighting of objects causing all the intensities to be recalculated and the object to be recoloured.

How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (2,0,0). System recalculates intensities and recolours moved object.

2. lightPos-invalid-outBounds

   Control: Automatic

   Input: New Point (11,0,0) for centre of light

   Output: Error Message "This light source is outside of the room. It cannot be rendered. Please enter a different location for this object.". Log message "Error: Tried to move light source to (11,0,0).

   Test Case Derivation: Scene size is defined to be (SCENE_HEIGHT, SCENE_WIDTH, SCENE_DEPTH); if any component of the light source's position is greater than any component of the scene size then the light source is out of bounds. Out of bounds light sources cannot light the scene.

   How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (11,0,0). System throws an error.

3. lightPos-invalid-onObj

   Same as *objPosition-invalid-onLight*.

4. lightPos-invalid-onObserver

   Control: Automatic

   Input: New Point (0,0,0) for centre of light source.

Output: Error Message “The centre of this light source intersects the observer. It cannot be rendered. Please enter a different location for this object.”. Log message “Error: Tried to move light source to observer position, (0,0,0)”.

Test Case Derivation: As per the requirement (**??** light sources cannot be on top of the observer. We impose that they cannot have their centres at the same position.

How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (0,0,0). System throws an error.

5. lightPos-valid-boundary

   Control: Automatic

   Input: New Point (0,10,0) for centre of light source.

   Output: Error Message “Parts of this light source are outside of the scene size. Please move the light and re-render the scene.”. Log Message “Error: Light source partially outside of room.”

   Test Case Derivation: Constraints on light position said $0 \ll x \leq SCENE\_HEIGHT$, which means that the centre is on the wall of the scene. This means part of the light would be rendered outside the scene.

   How test will be performed: Valid scene is loaded. Testing framework automatically assigns new position to (0,10,0). System throws an error.

6. lightPos-valid-besideObject

   Same as *objPosition-valid-besideLight*. Not to be tested because it covers the same scenario.

7. lightPos-valid-behindObject

   Same as *objPosition-valid-betweenLightAndViewer*. Not to be tested because it covers the same scenario.

8. lightColour-valid

Control: Automatic

Input: New (r,g,b) value for LIGHT_COLOUR picked from GUI picker = (10,255,50).

Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed to reflect the new light colour. All intensities need to be recalculated.

Test Case Derivation: The intensity of the LIGHT_COLOUR at a point is part of all lighting model calculations. Changes to the LIGHT_COLOUR requires recalculating all of the intensity values with this new (r,g,b) information.

How test will be performed: Valid scene is loaded. Testing framework automatically assigns new r,g,b to (10,255,50). System recalculates intensities and recolours object.

9. lightShape-valid

   Control: Automatic

   Input: New light type selection from dropdown list: directional.

   Output: *RENDERS_DIR/render-valid-AllInputs* with visuals changed to replace the point light with a directional light. Some incidence rays have changed, so all intensities need to be recalculated.

   Test Case Derivation: Different types of light project light rays differently. As such the system needs to adapt to the changing type of lights available

   How test will be performed: Valid scene is loaded. Testing framework automatically assigns new light type. System recalculates intensities and recolours new object.

[There is a great variety of test cases, and many exceptions are covered, but I still don't see how you can tell that the scene is rendered correctly with respect to the mathematical model? Do you have a pseudo-oracle that you can compare the results to? Can you do a bitwise comparison between your image and the "correct" image? If not, then you at least need some manual tests to verify that the scene renderings look correct. —SS]

5.2 Tests for Nonfunctional Requirements

The following section outlines the test cases for the non-functional requirements of the system. In particular these tests focus on the usability of the system, encompassing aspects like ease of installation, reliability, and learnability.

5.2.1 Usability

Ease of Installation The following test cases focuses on assessing the ease of installation of the system. It would be unrealistic to test all potential install environments; however due to this being implemented in Unity, the installation environments of this system are limited to those that Unity can run on.

For this type of testing we are testing to pass, not testing to fail - i.e. we want all of these test cases to pass. We are not concerned with cases of invalid input, since that would just mean not installing our software. We let the Unity error handling inform the user if the installation is unsuccessful.

1. install-clean-modern-win

   Type: Manual

   Initial State: Clean installation of Unity version CURRENT_VERSION on a Windows 10 machine.

   Input/Condition: Install system package.

   Output/Result: Installation of Unity version CURRENT_VERSION on a Windows 10 machine with our system installed.

   How test will be performed: The user will have a fresh install of Unity CURRENT_VERSION on a virtual windows machine. They will have a copy of the system package to import into Unity. The user will then be asked to fill out the Installability Evaluation borrowed from Smith et al. (2018) (See 1).

2. install-clean-modern-mac

   Type: Manual

Initial State: Clean installation of Unity version CURRENT_VERSION on a Mac OS machine.

Input/Condition: Install system package.

Output/Result: Installation of Unity version CURRENT_VERSION on a MacOS machine with our system installed.

How test will be performed: The user will have a fresh install of Unity CURRENT_VERSION on a virtual MacOS machine. They will have a copy of the system package to import into Unity. The user will then be asked to fill out the Installability Evaluation borrowed from Smith et al. (2018) (See 1).

3. install-clean-previous-windows

   Type: Manual

   Initial State: Clean installation of Unity version PREVIOUS_VERSION on a Windows 10 machine.

   Input/Condition: Install system package.

   Output/Result: Installation of Unity version PREVIOUS_VERSION on a Windows 10 machine with our system installed.

   How test will be performed: The user will have a fresh install of Unity PREVIOUS_VERSION on a virtual windows machine. They will have a copy of the system package to import into Unity. The user will then be asked to fill out the Installability Evaluation borrowed from Smith et al. (2018) (See 1).

4. install-clean-previous-mac

   Type: Manual

   Initial State: Clean installation of Unity version PREVIOUS_VERSION on a Mac OS machine.

   Input/Condition: Install system package.

   Output/Result: Installation of Unity version PREVIOUS_VERSION on a MacOS machine with our system installed.

How test will be performed: The user will have a fresh install of Unity PREVIOUS_VERSION on a virtual MacOS machine. They will have a copy of the system package to import into Unity. The user will then be asked to fill out the Installability Evaluation borrowed from Smith et al. (2018) (See 1).

5.2.2 Reliability

The following test cases focus on assessing the reliability of the system to repeatedly perform routine tasks. The purpose of these tests is to determine the load the system, and ensure that the output is reliable. These tests run the system through the functional test cases multiple times in succession.

1. reliable-loading

   Type: Automatic [How is this automated? Do you have a unit testing framework that can do this? —SS]

   Initial State: No scene loaded.

   Input/Condition: *SCENE_DIR/valid-AllInputs.JSON* (See 6.1)

   Output/Result: *SCENE_DIR/valid-AllInputs.JSON*

   How test will be performed: The system shall perform test *loadScene-allValid* RELIABILITY_THRESHOLD times in succession. The output should be the same as doing it once.

   [Interesting test case. It does sometimes confuse systems to do something like this twice. Good idea. —SS]

2. reliable-taskSwitch

   Type: Automatic

   Initial State: No scene loaded.

   Input/Condition: *SCENE_DIR/valid-AllInputs.JSON* (See 6.1), $k_a = 0.5$, $k_d = 0.5$, $k_s = 0.8$, $\alpha = 10$

   Output/Result: *SCENE_DIR/valid-AllInputs.JSON*

   How test will be performed: The system shall perform test *loadScene-allValid* and then in sequence make changes to the object properties. The output file should have the following values:

5.2.3 Learnability

The following test cases focus on understanding how easy the system is to learn for new users. In this case, we consider new users as those who are have less than 1 years experience in working with 3D graphics and lighting (i.e. Unity, or equivalent). The purpose of these tests is to see if a user with no background knowledge of these types of systems can perform basic operations successfully with no assistance.

The rationale is to see learnability as a ratio of mistakes made to task completion time. Mistakes in this case constitute errors and the amount of times they asked for assistance or sought ought documentation. The reason these are captured in mistakes is because our software ought to be intuitive to use; any time the user requires assistance it is because our software was unintuitive.

All learnability test cases share the following properties:

Initial State: No scene loaded.

1. learnability-loadScene

   Type: Manual

   Input/Condition: *SCENE_DIR/valid-AllInputs.JSON* (See 6.1)

   Output/Result: *SCENE_DIR/valid-AllInputs.JSON*. Time to completion. Number of errors. Number of times they sought assistance.

   How test will be performed: The user will have Unity open with the system installed. The user will proceed to load the input file to the system. Time will be measured from when the user opens Unity to when the output file is rendered. The number of errors will be measured as the number of misclicks made while performing this task. The number of times they sought assistance will me measured as times when they asked for help, or consulted any documentation. Users will also fill out the (6.4) or Expert (6.4) Usability Survey depending on their familiarity with other 3D graphics programs.

2. learnability-taskSequence

   Type: Manual

Input/Condition: *SCENE_DIR/valid-AllInputs.JSON*(See 6.1), $k_a = 0.5$, $k_d = 0.5$, $k_s = 0.8$, $\alpha = 10$.

Output/Result: *SCENE_DIR/valid-AllInputs.JSON*. Time to completion. Number of errors. Number of times they sought assistance.

How test will be performed: The user will have Unity open with the system installed. The user will proceed to load the input file to the system. The user will proceed to change the values of the object material properties to those given in the input. Time will be measured from when the user opens Unity to when the output file is rendered. The number of errors will be measured as the number of misclicks/mistypes made while performing this task. The number of times they sought assistance will me measured as times when they asked for help, or consulted any documentation. Users will also fill out the Novice (6.4) or Expert (6.4) Usability Survey depending on their familiarity with other 3D graphics programs.

[The learnability measurements sound good to me. I hope you have time to at least do a few measurements. —SS]

5.3 Traceability Between Test Cases and Requirements

The following section summarizes the relationships between test cases and requirements. We have restated the requirements originally laid out in the CA here for convenience. Note that requirements 12-14 are not covered by test cases - this is because their requirements are not part of the objective of this test plan.

Requirements from CA [For maintainability reasons, it usually isn't a good idea to copy and paste between documents. A cross-reference to the CA document should be all that you need. —SS]

R1: When presented with a scene in a file, the system shall correctly read from file the input data for light source(s) and object(s).

R2: System responds with specific error message when system cannot read input files.

R3: The system asks user if they would like to use the default settings when scene size, shading model, and/or reflection model information is missing from input, and applies (DEF_HEIGHT, DEF_WIDTH, DEF_DEPTH), DEF_SHADE and/or DEF_LIGHT if the user answers yes.

R4: When no input file is given, the system provides a default scene of dimension (DEF_HEIGHT, DEF_WIDTH, DEF_DEPTH) with one point light source with the default light colour, one sphere with the default material properties, rendered using default shading (DEF_SHADE) and lighting (DEF_LIGHT).

R5: The system shall verify that all input data meets constraints laid out in the CA.

R6: The system responds with specific error message when user inputs contain errors (type mismatch, data outside of constraints).

R7: The library shall correctly calculate the surface normals for object(s) based on shading model.

R8: The library shall calculate the incidence and reflection vectors off of object(s) surface(s) based on light position(s), object(s) properties, shading model and observer position.

R9: The library shall calculate the light intensity based on light position(s), object(s) material properties, and shading model.

R10: The library shall calculate the final colour object(s) faces based on the intensities calculated in R9.

R11: The library will output code for a lit and shaded scene.

R12: Users can render a default scene (define in R4) faster than in OpenGL.

R13: The addition of new input methods should not affect the usability of the system.

R14: The addition of new lighting models, shading models, types of light sources and/or types of objects should be completable in MODIFICATION_TIME_THRESHOLD.

R15: USABILITY_THRESHOLD % of users can install the system without requiring assistance.

R16: USABILITY_THRESHOLD % of users can load an existing scene with no assistance.

R17: USABILITY_THRESHOLD % of users can change the parameters of the lighting models and re-render an existing scene with no assistance.

R18: USABILITY_THRESHOLD % of users can initialise a new scene with the default parameters (default object, light source, lighting model, and shader) with no assistance.

R19: USABILITY_THRESHOLD % of users perceive Lights, Camera, Models!to be easier to use than OpenGL.

R20: USABILITY_THRESHOLD % of users perceive Lights, Camera, Models!to allow them more control than the built in Unity shader options.

	Functional Requirements										
Test Case	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**
loadScene-allValid	X				X		X	X	X	X	X
loadScene-validMissingSome	X		X		X		X	X	X	X	X
loadScene-fileExistNoData	X			X	X		X	X	X	X	X
loadScene-invalidInput		X			X	X					
createDefault-validMissingData			X				X	X	X	X	X
createDefault-fileExistNoData				X			X	X	X	X	X
lightModel-valid	X				X				X	X	X
shadingModel-valid	X				X		X		X	X	X
objMaterialPropChange-valid-ks					X				X	X	X
objMaterialPropChange-valid-kd					X				X	X	X
objMaterialPropChange-valid-ka					X				X	X	X
objMaterialPropChange-valid-α					X				X	X	X
objMaterialPropChange-invalid-ks					X	X			X	X	X
objMaterialPropChange-invalid-kd					X	X			X	X	X
objMaterialPropChange-invalid-ka					X	X			X	X	X
objMaterialPropChange-invalid-α					X	X			X	X	X

	Functional Requirements										
Test Case	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**
objMaterialPropChange-bound-ks					X				X	X	X
objMaterialPropChange-bound-kd					X				X	X	X
objMaterialPropChange-bound-ka					X				X	X	X
objMaterialPropChange-bound-α					X				X	X	X
objPosition-valid					X		X	X	X	X	X
objPosition-invalid-outBounds					X	X	X	X	X	X	X
objPosition-invalid-onLight					X	X	X	X	X	X	X
objPosition-invalid-onObserver					X	X	X	X	X	X	X
objPosition-valid-edgeOfRoom					X		X	X	X	X	X
objPosition-valid-betweenLightAndViewer					X		X	X	X	X	X
objPosition-valid-besideLight					X		X	X	X	X	X
objColour-valid-base					X				X	X	X
objColour-valid-specular					X				X	X	X
objShape-valid					X				X	X	X
lightPos-valid					X		X	X	X	X	X
lightPos-invalid-outBounds					X	X	X	X	X	X	X
lightPos-invalid-onObj					X	X	X	X	X	X	X
lightPos-invalid-onObserver					X	X	X	X	X	X	X
lightPos-valid-boundary					X		X	X	X	X	X
objPosition-valid-behindObject					X		X	X	X	X	X
objPosition-valid-besideObject					X		X	X	X	X	X
lightColour-valid					X		X	X	X	X	X
lightShape-valid					X		X	X	X	X	X

	Non-Functional Requirements								
Test Case	**R12**	**R13**	**R14**	**R15**	**R16**	**R17**	**R18**	**R19**	**R20**
loadScene-validMissingSome							X		
loadScene-fileExistNoData							X		
install-clean-modern-win				X					
install-clean-modern-mac				X					
install-clean-previous-win				X					
install-clean-previous-mac				X					
reliable-loading					X				
reliable-taskSwitch					X	X			
learnability-loadScene					X			X	X
learnability-taskSequence					X	X		X	X

References

Spencer Smith, Zheng Zeng, and Jacques Carette. Seismology software: state of the practice. *Journal of Seismology*, 22, 02 2018. doi: 10.1007/s10950-018-9731-3.

6 Appendix

6.1 Input Files

This section contains the contents of the input files used for the test cases.

valid-AllInputs.json

```
{
        "Height" : 50,
        "Width" : 50,
        "Depth" : 50,

        "Object":[
                {
                        "type": "sphere",
                        "position": [10,0,10],
                        "size": 1,
                        "ka": 1,
                        "kd": 1,
                        "ks": 0.5,
                        "alpha": 1,
                        "base_colour": [35,74,35],
                        "spec_colour": [255,255,255],
                }
        ],

        "LightSource": [
                {
                        "type": "point",
                        "position": [5,10,0],
                        "light_colour": [255,255,255],
                        "intensity": [10,10,10],
                }
        ],

        "Observer" : [
                {
                    "position" : [0,0,0],
                    "direction" : [1,0,1]
                }
        ],

        "ShadingModel" : "Gouraud",
        "LightingModel" : "Phong"
}
```

valid-MissingData.json

```
{
        "Height" : 50,
        "Width" : 50,
        "Depth" : 50,

        "Object":[
                {
                        "type": "sphere",
                        "position": [10,0,10],
                        "size": 1,
                        "ka": 1,
                        "kd": 1,
                        "ks": 0.5,
                        "alpha": 1,
                        "base_colour": [35,74,35],
                        "spec_colour": [255,255,255],
                }
        ],

        "LightSource": [
                {
                        "type": "point",
                        "position": [5,10,0],
                        "light_colour": [255,255,255],
                        "intensity": [10,10,10],
                }
        ],

        "Observer" : [
                {
                        "position" : [0,0,0],
                        "direction" : [1,0,1]
                }
        ],

        "ShadingModel" : ,
        "LightingModel" :
}
```

valid-NoData.json

```
{

}
```

valid-invalid.json

```
{
        "Height" : 50,
        "Width" : 50,
        "Depth" : 50,

        "Object":[
                {
                        "type": "sphere",
                        "position": [10,0,10],
                        "size": 1,
                        "ka": 2,
                        "kd": 1,
                        "ks": 0.5,
                        "alpha": 1,
                        "base_colour": [35,74,35],
                        "spec_colour": [255,255,255],
                }
            ],

        "LightSource": [
                {
                        "type": "point",
                        "position": [5,10,0],
                        "light_colour": [255,255,255],
                        "intensity": [10,10,10],
                }
        ],

        "Observer" : [
                {
                        "position" : [0,0,0],
                        "direction" : [1,0,1]
                }
        ],

        "ShadingModel" : "Flat",
        "LightingModel" : "Lambertian"
}
```

6.2 Installability Evaluation Smith et al. (2018)

6.3 Symbolic Parameters

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

Installability(Measured via installation on a virtual machine.)	
Are there installation instructions?	(yes, no)
Are the installation instructions linear?	(yes, no, n/a)
Is there something in place to automate the installation?	(yes, no)
Is there a specified way to validate the installation, such as a test suite?	(yes, no)
How many steps were involved in the installation?	(number)
How many software packages need to be installed before or during installation?	(number)
(I) Run uninstall, if available. Were any obvious problems caused?	(unavail, yes, no)
Overall impression?	(1...10)

Table 1: Installability Evaluation from Smith et al. (2018)

Symbolic Constant	**Value**
DEF_HEIGHT	10
DEF_WIDTH	10
DEF_DEPTH	10
DEF_SHADE	Gouraud
DEF_LIGHT	Lambertian
PREVIOUS_VERSION	2018.2.16
CURRENT_VERSION	2019.2.11
RELIABILITY_THRESHOLD	90%
USABILITY_THRESHOLD	90%
MODIFICATION_TIME_THRESHOLD	2 DAYS

6.4 Usability Survey Questions?

The following section outlines usability surveys for the system. It is split up into two surveys: for novices, and for experts. The rationale for splitting the survey up is because of the different expectations each user group will have for the software. Novice users tend to focus on completing the task, and the learnability of the software. Expert users tend to focus on task efficiency, and the extendability of the software.

Novice User Usability Survey

Question	**Rationale**
On a scale of 1 (extremely difficult) to 5 (extremely easy), how easy was the installation of the system into Unity?	To measure the perceived ease of installation; the installability evaluation grades the ease of installation documentation and process (through number of steps) without accounting for the user base's existing experience with installing software. For example, an installation with 5 large steps may be more difficult to follow than one with 10 small steps. The installation process should be intuitive enough that the user doesn't need documentation.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that you could load a scene in this system?	To quantify learnability (and retention) of basic function — loading a scene.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that you could change the material parameters of an object in this system?	To quantify learnability (and retention) of basic function — modifying an object's properties.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that you could change the position of an object in this system?	To quantify learnability (and retention) of basic function — modifying an object's position.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that you could change the parameters of light source in this system?	To quantify learnability (and retention) of basic function — modifying a light source.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that the system will output a correctly rendered scene?	To quantify trust between the user and the system — effectively measuring the perceived reliability.
On a scale of 1 (extremely unlikely) to 5 (extremely likely), how likely are you to use this system for local illumination?	To judge whether the system perceivably satisfies their needs.

Expert User Usability Survey

Question	**Rationale**
On a scale of 1 (extremely difficult) to 5 (extremely easy), how easy was the installation of the system into Unity?	To measure the perceived ease of installation; the installability evaluation grades the ease of installation documentation and process (through number of steps) without accounting for the user base's existing experience with installing software. For example, an installation with 5 large steps may be more difficult to follow than one with 10 small steps. The installation process should be intuitive enough that the user doesn't need documentation.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that you could load a scene in this system?	To quantify learnability (and retention) of basic function — loading a scene.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that you could change the material parameters of an object in this system?	To quantify learnability (and retention) of basic function — modifying an object's properties.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that you could change the position of an object in this system?	To quantify learnability (and retention) of basic function — modifying an object's position.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that you could change the parameters of light source in this system?	To quantify learnability (and retention) of basic function — modifying a light source.
On a scale of 1 (not at all confident) to 5 (extremely confident), how confident are you that the system will output a correctly rendered scene?	To quantify trust between the user and the system — effectively measuring the perceived reliability.
Which system was easiest to use for loading a scene (circle one): Unity, System, OpenGL	The point is to have them single out preferences against the competition for this specific task.
Which system was easiest to use for changing the properties of an object (circle one): Unity, System, OpenGL	The point is to have them single out preferences against the competition for this specific task.
Which system was easiest to use for changing the properties of light source (circle one): Unity, System, OpenGL	The point is to have them single out preferences against the competition for this specific task.

Expert User Usability Survey (continued)

Question	**Rationale**
Which system was easiest to create a scene for (circle one): Unity, System, OpenGL	The point is to have them single out preferences against the competition for this specific task.
Which system handles the most lighting options (circle one): OpenGL, Unity, System.	The showcase which system is considered the most extendable.
On a scale of 1 (extremely unlikely) to 5 (extremely likely), how likely are you to use this system for local illumination?	To judge whether the system perceivably satisfies their needs.