Project Title: Unit Verification and Validation Plan for ProgName

Author Name

December 2, 2019

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

Contents

List of Tables

[Do not include if not relevant —SS]

List of Figures

[Do not include if not relevant —SS]

2 Symbols, Abbreviations and Acronyms

symbol	description
T	Test

[symbols, abbreviations or acronyms – you can reference the SRS, MG or MIS tables if needed —SS]

This document outlines the unit testing plan for ProgName. The purpose of this is to identify the test cases that address the functionality of the modules for this program. This documentation alongside the System VnV Test Plan comprise the testing of the entire ProgNamesystem.

In this document I will first explain outline some general information about the document, including the software that is being tested and the scope of this document (i.e. the modules that this covers). This will be followed by an outline of the testing procedure including the testing team, and the automated testing tools and methods used. I then cover the unit test descriptions, rationale, and relationship to the MIS.

3 General Information

3.1 Purpose

The purpose of this document is to outline test cases for the modules of the ProgNamesystem. The ProgNamesystem implements several shading and lighting models and renders it in the Unity 3D environment. More information about this software can be found in its associated documentation at : [insert github url].

3.2 Scope

The scope of this document covers the modules outlined in the MIS.

Of the modules covered, the following will not be implemented by me for this project:

- Rendering Module **??**
- JSON Module **??**
- Hardware Hiding Module **??**

These modules are not going to be implemented by me for this work as they exist integrated into the larger Unity system that this software is being created in. It's not anticipated that this development/system environment will change between versions of this software, so it makes sense to use the built-in services for these and use Unity's existing API for them. As these

modules' services exist as part of the Unity development environment I will not be testing their functioning. I assume that their behaviour is correct as shipped with Unity.

As I am implementing all of the other modules they will need to be tested. I have prioritized their testing in the following order:

1. Point3D module **??**
2. Vector module **??**
3. Vector Math module **??**
4.

Several of the modules that I am choosing to implement also exist natively in the Unity environment. The reasoning for choosing to implement them instead of using the built-in modules are to two-fold; firstly, I do this to improve the information hiding element of the system design. By putting personal wrappers around the built-in functionality it future proofs the system against changes in the Unity API, as maintenance will only rely on changing API calls in the wrappers and not throughout the program. Secondly, it draws a specific parallel between the designed documents and the actual implemented system. This way designers reading the documentation can easily see where specific functionality is coming from and track the design decisions made in these documents all the way to the code.

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 validation 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.
Dr. Spencer Smith	Secondary Reviewer	Ensure reasonable coverage of design considerations and requirements as part of marking these documents.

4.2 Automated Testing and Verification Tools

Unit testing for this system will use the Unity Test Framework (UTF) https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html?_ga=2.163129468.110989482.1575244303-869222737.1569978243 This is the built in automated unit and integration testing suite for Unity. This was chosen for the simplicity of use in this environment; there are some limitations to this framework as outlined in their documentation. The results of the unit and integration tests are visually displayed in Unity when tests are run; passed tests have a green check next to their name, and failed tests have a red cross.

As the recommended programming environment for Unity is Visual Studio, I will leverage the tools shipped with this IDE to perform code coverage analysis to ensure that there is no dead code in the system. The purpose of checking for dead code in this system is to make sure that all the planned functionality is actually possible.

4.3 Non-Testing Based Verification

There will be no non-testing based verification for this project. This is because the automatic testing should discover the code-based functionality issues. At that point behavioural issues with the software system are likely to be a fault in the implementation design. These types of issues will be caught in the System Tests, and so those details are outlined in the System VnV Plan **??**.

5 Unit Test Description

These unit tests are designed around modules listed in the MIS (**??**).

5.1 Tests for Functional Requirements

The following sections describe the unit tests that will be run for each module in this system. Many of these tests will deal with providing the same function several values that test its response/behaviour to "bad" data. To increase the readability of this document, the tests for each function will only be documented once, with the various values passed to it summarized in tabular form.

5.1.1 Point3D Module

The following section outlines tests for the Point3D module. As this module is an abstract data type with very basic functionality the tests for this section will be simple. Getter function tests will not be documented here as they will be assumed correct in their functionality.

1. CreatePoint

   Type: Functional Automatic

   Initial State: –

   Input:

Variation Name	Passed Values
Valid Values	x:= 10, y:= 10, z:= 10
Invalid Values	x:= A, y:= B, z:= C
Empty Values	x:= null, y:= null, z:= null

   Output:

Variation Name	Expected Output
Valid Values	–
Invalid Values	Exception:= INVALID_POINT
Empty Values	Exception:= INVALID_EMPTY_POINT

   Test Case Derivation: The purpose of this function is to create a 3D cartesian point in line with the Point() access program. At this point there is no error checking in the system for the placement of the point

relative to anything else, so all exceptions and tests in its creation only check that the input matches the function description. There is no test variation for duplication of points as that's currently allowed in the system.

How test will be performed: The test will be handled automatically by test script through the Unity Testing Framework. The tests will be coded as a C# file that will be read by the TestRunner API.

2. FindDistance

Type: Functional Automatic

Initial State: Point3D has been successfully created with the following position P = Point(1,1,1).

Input:

Variation Name	**Passed Values**
P == Q	Q := new Point(x:= 1, y:= 1, z:= 1)
P is far from Q	Q := new Point(x:= 100, y:= 100, z:= 100)
P is close to Q	Q := new Point(x:= -0.1, y:= 0.1, z:= -0.1)

Output:

Variation Name	**Expected Output**
P == Q	out:= 0
P is far from Q	out:= 171.4730299493189
P is close to Q	out:= 1.905255888325765

Test Case Derivation: The purpose of this test case is to test the accuracy of this calculation. The variations in the context of two points in space can be categorized as either they are the same point (same position), they are close to each other, and they are far away from each other. For the last two variations (close and far) boundary test cases would be the particular integer or floating point bounds to test for underflow and overflow.

We don't consider the test case that a bad point Q is passed to this function because the bad value cases will be caught in the constructor tests.

How test will be performed: The test will be handled automatically by test script through the Unity Testing Framework. The tests will be coded as a C# file that will be read by the TestRunner API.

5.1.2 Vector Module

The following section outlines tests for the Vector module. As this module is an abstract data type with very basic functionality the tests for this section will be simple. Getter function tests will not be documented here as they will be assumed correct in their functionality.

1. CreateVector-Points
   The following test case outlines the creation of a vector from two points, P and Q, which represent the start and end of that vector.

   Type: Functional Automatic

   Initial State: –

   Input:

Variation Name	Passed Values
P == Q	P:= Point(1,1,1), Q:= Point(1,1,1)
Valid Values	P:= Point(1,1,1), Q:= (0,1,-1)

   Output:

Variation Name	Expected Output

   Test Case Derivation:

   How test will be performed: The test will be handled automatically by test script through the Unity Testing Framework. The tests will be coded as a C# file that will be read by the TestRunner API.

2. CreateVector-DM

   Type: Functional Automatic

   Initial State: Input:

Variation Name	Passed Values

   Output:

Variation Name	Expected Output

   Test Case Derivation:

   How test will be performed: The test will be handled automatically by test script through the Unity Testing Framework. The tests will be coded as a C# file that will be read by the TestRunner API.

5.1.3 Vector Math Module

The following section outlines tests for the Vector Math module. As this module is an library of calculations, the functionality is very explicitly defined. For these test cases the results are checked against the mathematical results of the answer.

1. Addition
   The following test case outlines the addition of two Vectors. The output of all of these variations will be a new Vector that represents the addition of the two input Vectors. For these cases the creation of the new Vector is not checked as it is assumed to be covered by the testing of the Vector module.

   Type: Functional Automatic

   Initial State: –

   Input:

Variation Name	**Passed Values**
V1 == -V2	V1:= Vector(Point(0,0,0),Point(1,1,1)), V2:= Vector(Point(1,1,1),Point(0,0,0))
V1 == V2	V1:= Vector(Point(0,0,0),Point(1,1,1)), V2:= Vector(Point(0,0,0),Point(1,1,1))
V1 ≠ V2	V1:= Vector(Point(1,2,4),Point(1,1,1)), V2:= Vector(Point(10,5,0),Point(1,3,2))

   Output:

Variation Name	**Expected Output**
V1 == -V2	out:= Vector(0,0,0,m:=0)
V1 == V2	out:= Vector(2,2,2,m:=3.464101615137755)
V1 ≠ V2	out:= Vector(ux,uy,uz,m)

   Test Case Derivation:

   How test will be performed: The test will be handled automatically by test script through the Unity Testing Framework. The tests will be coded as a C# file that will be read by the TestRunner API.

2. ScalarMultiply

   Type: Functional Automatic

Initial State: –

Input:

Variation Name	Passed Values
Valid Values	V:= Vector(1,1,1,m:=1.7320508), k:=3
Valid Values Negative	V:= Vector(1,1,1,m:=1.7320508), k:=-3
Valid Values Zero	V:= Vector(1,1,1,m:=1.7320508), k:=0

Output:

Variation Name	Expected Output
Valid Values	V_{out}:= Vector(3,3,3,m:=5.1961524)
Valid Values Negative	V_{out}:= Vector(-3,-3,-3,m:=5.1961524)
Valid Values Zero	V_{out}:= Vector(0,0,0,m:=0)

Test Case Derivation:

How test will be performed: The test will be handled automatically by test script through the Unity Testing Framework. The tests will be coded as a C# file that will be read by the TestRunner API.

5.1.4 Scene Module

The following section outlines tests for the Scene module. This module is an abstract data type that acts as a control module for the rest of the system. Getters for this module are not tested/documented here. The following test cases rely on other modules having been tested and working correctly; as such this module will not check for valid construction of other objects.

1. CreateScene
   The following test case outlines the creation of a Scene. In the system the initScene program is populated with information from the Input-Parameters module.

   Type: Functional Automatic

   Initial State: –

Input:

Variation Name	Passed Values
Valid Values	height:= width:= depth:= obs:= Observer() ls:= LightSource() os:= Object() lightModel:= DIFFUSE

Output:

Variation Name	Expected Output

Test Case Derivation:

How test will be performed: The test will be handled automatically by test script through the Unity Testing Framework. The tests will be coded as a C# file that will be read by the TestRunner API.

2. ChangeScene
   The following test cases outline changes made to the Scene. This captures the functionality of several access programs but they all result in Scene transitions. Type: Functional Automatic

   Initial State: – Input:

Variation Name	Passed Values

   Output:

Variation Name	Expected Output

   Test Case Derivation:

   How test will be performed: The test will be handled automatically by test script through the Unity Testing Framework. The tests will be coded as a C# file that will be read by the TestRunner API.

5.2 Tests for Nonfunctional Requirements

[If there is a module that needs to be independently assessed for performance, those test cases can go here. In some projects, planning for nonfunctional tests of units will not be that relevant. —SS]

[These tests may involve collecting performance data from previously mentioned functional tests. —SS]

5.2.1 Module ?

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   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 Module ?

...

5.3 Traceability Between Test Cases and Modules

[Provide evidence that all of the modules have been considered. —SS]

References

6 Appendix

[This is where you can place additional information, as appropriate —SS]

6.1 Symbolic Parameters

[The definition of the test cases may call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance. —SS]