A Celestia Testing Framework

By: Team Go-Gitters
Jacob Mattox, Kyle Cooper, Lara Brooksbank, Alexander Swanson

CSCI 362

Table of Contents:

Go-Gitters Deliverable 1: Celestia

Report:

We chose the space-based project Celestia. Cloning the project from Github was not difficult, however we immediately encountered problems when trying to build Celestia. While it seemed to be documented well, there was almost zero help with the actual build from the source code. There is an ‘Install’ file that gives a very basic breakdown of the process for getting ./configure and make to work, but it leaves out many of the finer details such as all the libraries required and how to make the configure file visible. Even when downloading the specific required dependencies we had trouble figuring out the correct versions, such as developer vs. community. We understand that some of this should be common knowledge, but beginners often struggle catching up because there is too much assumed knowledge in a project. Once we got the correct dependencies downloaded and ran the make command we were still getting errors because Celestia uses the Lua scripting language for scripts, and although Lua is installed with the original download, for some reason you have to explicitly specify the file path for Lua within the Celestia directory. All these combined issues were a real headache when just trying to build the original executable.

We have still been unable to run any test cases. There are a few folders with the test name, but there are no clear indications of what the test is or how to execute them. Instead of executing the test cases, we created a guide to install the dependencies for Celestia more easily. Additionally, Celestia seems to have a large online forum for users and developers to interact. This could be a boon to the project, but it is a massive online repo of information for a new person to go through and takes a lot of time. This is not a bad thing inherently, but is definitely something to new user should know before diving in.

At this point in the project, we all agree that although Celestia seems like a really interesting project to work on, we did not fully understand the scope of our endeavor before choosing. Each of us has been more than a little frustrated so far with the barrier of entry into basic understanding seemingly way above where any of us thought it would be. Hopefully, as we continue learning the system, we can expand our knowledge and skill base and not feel so overwhelmed with this project.

Go-Gitters Deliverable 2: Celestia

Test plan for Celestia:

Testing Process:

Celestia is primarily a graphical application that creates a replica of our universe. Our testing process will involve interacting with the GUI via Python and Bash scripting. The Python library PyAutoGui allows for the system to take control of the keyboard and mouse functions and use them to navigate the screen using pixels for reference. We are leveraging this library to test what Celestia displays after certain navigational functions and scripts are run.

Requirements Traceability:

Celestia is an open-source project that was not designed by our group and therefore we do not have direct access to the requirements of its design. However, we can infer some requirements based on the existing product. It would appear that a requirement is for the user to be able to interact with the universe in a tangible way. Another requirement could be to help the user learn more about our solar system. Since these are the requirements we can infer based on our use of system, these are the types of things we will be testing.

Tested Items:

We are testing the installation process which, to a new user, could seem complex and create a barrier to entry for using the system. Additional tests will be of existing functions within the Celestia packages, as well as, the GUI.

Testing Schedule:

10/15: Deliverable 2 - 5 test cases created
11/12: Deliverable 3 - Initial test framework with 5 tests created and demonstrated
11/19: Deliverable 4 - Test framework with 25 tests finalized and demonstrated
11/28: Deliverable 5 - 5 faults injected into codebase to create failed test cases, demonstrated
11/30: Project completed and demonstrated again, with final thoughts on complete process

Test Recording Procedure:

Our test cases are documented in a test log text file that lists the date and time of the test and the output, whether a success or failure, in a readable format. Due to the inherent nature of our program (being mostly graphical), a large portion of our test cases will only return “Success” if the test script has been executed completely. Any failure would most likely cause a failure to the entire program and would require a restart.

Hardware and Software Requirements:

Operating System: All code was tested in Linux using the Ubuntu 18.04 distribution

Hardware: All tests performed on a 13” MacBook Pro within a VirtualBox virtual environment. Graphical tests use the MacBook Pro’s native resolution for pixel density to find the specified location on the screen.

Software: VirtualBox, Python, Bash, Ubuntu 16.04, PyAutoGui(python library)

Constraints:

Time is a major constraint on this project. We only have one semester and each of us have other classes and assignments to complete during the semester. Total number of people is also a concern. There are four people working on the project, but as with any testing, the more people the better for creating useful test cases.

System Tests:

5 Test Cases

Test 1: Installation - When initially building Celestia in Ubuntu 18.04, test for the necessary libraries required to run ‘./configure --with-gtk’.

Test 2: Installation - When initially building Celestia in Ubuntu 18.04, test for the appearance of the ‘Make’ file after running the ./Configure command to compile from Celestia source code.

Test 3: Installation - After running ‘sudo make install’, test for the appearance of the celestia.exe file.

Test 4: Test that Celestia graphical interface properly fixes onto the sun using hotkeys.

Test 5: Test that Celestia graphical interface properly zooms out to show the Milky Way galaxy.

Report:

Thus far, getting Celestia to compile in our virtual machines has been the most difficult and time consuming part of the project. In the Celestia Github and Celestia Forums there is no support for compiling Celestia from the source code in Linux/Ubuntu 18.04. We decided as a team that this would be a worthwhile place to begin before diving into the rest of Celestia itself. Our first few test cases revolve around compiling and installing Celestia from the official source code. These tests include checking for the necessary libraries, the location and appearance of the 'Make' file, and also the location and appearance of the celestia.exe after compiling.

Go-Gitters Deliverable 3: Celestia

1. Architecture:

The testing framework for Celestia was built in the following structure:

- ➔ TestAutomation - Holds all other directories in the framework
  - ◆ Celestia - Holds the Celestia project files cloned from github
  - ◆ docs - Holds the README.md
  - ◆ oracles - Not used but required in the framework
  - ◆ reports - Not used but required in the framework
  - ◆ scripts - Holds all scripts required for running the framework (one script for the user to run and helper scripts for the execution)
  - ◆ temp - Holds the text files that are output of scripts and an html file that is created and opened automatically to display results
  - ◆ testCases - Not used but required in the framework
  - ◆ testCasesExecutables - Not used but required in the framework

2. How To Run:

This framework was designed on Ubuntu 18.4 distribuition of Linux and as such has Linux specific commands nested throughout. The test framework itself does not need a full build of the project but it does check for needed packages and installs them as necessary.

From your command line type the following:

git clone --recursive https://github.com/csci-362-fall-2018-01/Team-Go-Gitters.git
(This will recursively gather Celestia as a submodule)

cd Team-Go-Gitters/TestAutomation (Move to correct directory)

./scripts/runAllTests.sh (Starts the automated framework and executes an html with output)

3. 5 Test Cases:

3.1. Test 1:
- 3.1.1. Since Celestia was such a difficult project for us to build we decided that automating the check for the correct packages needed for the build.

3.2. Test 2:
- 3.2.1. Likewise this test makes sure the makefile required to build the project is found after executing the cmake (Celestia devs changed this on 11/26 - it previously configured in a different way)

3.3. Test 3:
- 3.3.1. Method Tested: degToRad()
- 3.3.2. Input: 720.0
- 3.3.3. Expected Output: 12.5663706144
- 3.3.4. Actual Output: 12.5663706144

3.4. Test 4:
- 3.4.1. Method Tested: degToRad()
- 3.4.2. Input: 0.0
- 3.4.3. Expected Output: 0.0
- 3.4.4. Actual Output: 0.0

3.5. Test 5:
- 3.5.1. Method tested: degToRad()
- 3.5.2. Input: -180.0
- 3.5.3. Expected Output: -3.14159265358979323846
- 3.5.4. Actual Output: -3.14159265358979323846

4. Report:

Our initial desire with Celestia was to test some of the graphical aspects of the project using a Python library that give you control over the GUI. At this point we are unable to get a functional test using the graphics because of the complexity for interacting with a GUI using only a script. Once we determined that it was less feasible to continue trying to make the graphics work, we decided that testing math functions would be the most helpful. Given that Celestia is a space exploration program, it heavily relies on its mathematical components to be correct.

From a team perspective, we are struggling to find time to really devote to the project. Without firm instructions from an authority, we have a hard time determining how long a task should take, and we therefore spend many hours on things we should probably walk away from. Working with a group of peers is difficult when no one takes the initiative to lead the group.

Go-Gitters Deliverable 4: Celestia

Overview:

For this deliverable, we continue with the creation of test cases by hashing out the remaining twenty. All test cases are listed below:

Test Cases:

1. Test 1:
    1.1. Since Celestia was such a difficult project for us to build we decided that automating the check for the correct packages needed for the build.

2. Test 2:
    2.1. Likewise this test makes sure the makefile required to build the project is found after executing the cmake (Celestia devs changed this on 11/26 - it previously configured in a different way)

3. Test 3:
    3.1. Method Tested: degToRad()
    3.2. Input: 720.0
    3.3. Expected Output: 12.5663706144
    3.4. Actual Output: 12.5663706144

4. Test 4:
    4.1. Method Tested: degToRad()
    4.2. Input: 0.0
    4.3. Expected Output: 0.0
    4.4. Actual Output: 0.0

5. Test 5:
    5.1. Method tested: degToRad()
    5.2. Input: -180.0
    5.3. Expected Output: -3.14159265358979323846
    5.4. Actual Output: -3.14159265358979323846

6. Test 6:
    6.1. Method tested: circleArea()
    6.2. Input: 6.0
    6.3. Expected Output: 113.097335529
    6.4. Actual Output: 113.097335529

7. Test 7:
    7.1. Method tested: circleArea()
    7.2. Input: 0.0
    7.3. Expected Output: 0.0
    7.4. Actual Output: 0.0

8. Test 8:
    8.1. Method tested: circleArea()
    8.2. Input: -1.0
    8.3. Expected Output: Error
    8.4. Actual Output: Error

9. Test 9:
    9.1. Method tested: radToDeg()
    9.2. Input: 3.14159265358979323846
    9.3. Expected Output: 180.0
    9.4. Actual Output: 180.0

10. Test 10:
    10.1. Method tested: radToDeg()
    10.2. Input: 0.0
    10.3. Expected Output: 0.0
    10.4. Actual Output: 0.0

11. Test 11:
    11.1. Method tested: radToDeg()
    11.2. Input: -2 * 3.14159265358979323846
    11.3. Expected Output: -360.0
    11.4. Actual Output: -360.0

12. Test 12:
    12.1. Method tested: square()
    12.2. Input: -1.0
    12.3. Expected Output: 1
    12.4. Actual Output: 1

13. Test 13:
    13.1. Method tested: square()
    13.2. Input: 0.0
    13.3. Expected Output: 0
    13.4. Actual Output: 0

14. Test 14:
    14.1. Method tested: square()
    14.2. Input: 53.0
    14.3. Expected Output: 2809
    14.4. Actual Output: 2809

15. Test 15:
    15.1. Method tested: clamp()
    15.2. Input: -2.0
    15.3. Expected Output: 0
    15.4. Actual Output: 0

16. Test 16:
    16.1. Method tested: clamp()
    16.2. Input: 0.0
    16.3. Expected Output: 0
    16.4. Actual Output: 0

17. Test 17:
    17.1. Method tested: clamp()
    17.2. Input: 0.5
    17.3. Expected Output: 0.5
    17.4. Actual Output: 0.5

18. Test 18:
    18.1. Method tested: cube()
    18.2. Input: -2.0
    18.3. Expected Output: -8.0
    18.4. Actual Output: -8.0

19. Test 19:
    19.1. Method tested: cube()
    19.2. Input: 0.0
    19.3. Expected Output: 0.0
    19.4. Actual Output: 0.0

20. Test 20:
    20.1. Method tested: cube()
    20.2. Input: 10.5
    20.3. Expected Output: 1157.625
    20.4. Actual Output: 1157.625

21. Test 21:
    21.1. Method tested: sign()
    21.2. Input: -2.0
    21.3. Expected Output: -1.0
    21.4. Actual Output: -1.0

22. Test 22:
    22.1. Method tested: sign()
    22.2. Input: 0.0
    22.3. Expected Output: 0.0
    22.4. Actual Output: 0.0

23. Test 23:
    23.1. Method tested: sign()
    23.2. Input: 5.5
    23.3. Expected Output: 1.0
    23.4. Actual Output: 1.0
24. Test 24:
    24.1. Method tested: sphereArea()
    24.2. Input: 10.0
    24.3. Expected Output: 1256.63706144
    24.4. Actual Output: 1256.63706144

25. Test 25:
    25.1. Method tested: sphereArea()
    25.2. Input: 0.0
    25.3. Expected Output: 0.0
    25.4. Actual Output: 0.0

26. Test 26:
    26.1. Method tested: sphereArea()
    26.2. Input: -5.0
    26.3. Expected Output: Error
    26.4. Actual Output: Error

27. Test 27:
    27.1. Method tested: pfmod()
    27.2. Input: 10.0, 5.0
    27.3. Expected Output: 0
    27.4. Actual Output: 0

28. Test 28:
    28.1. Method tested: pfmod()
    28.2. Input: 1.0, 2.0
    28.3. Expected Output: 1.0
    28.4. Actual Output: 1.0

29. Test 29:
    29.1. Method tested: pfmod()
    29.2. Input: 17.0, 0.0
    29.3. Expected Output: nan
    29.4. Actual Output: nan

Report:

This was a very challenging deliverable for our team. None of us have worked with C++ before and we struggled to find code to test that we understood. It was even more difficult to implement the code that we did find in a way that was repeatable with a script. Eventually we began to learn enough C++ to get our test code pulled out of the original source files and into a main function to test from. Getting the output to an html file proved to be challenging also, but we were able to combine similar cases into individual files and run “make” on them to more easily create the necessary executables while also keeping the access to output files.

Overall, this assignment has been extremely time consuming and we have not done a great job of managing our time effeciently. We have struggled to meet every deadline, and our project was a mess until recently. On top of all the problems we have internally as a group, externally the Celestia development team pushed a good many changes to the repository that changed many things about the install and build of Celestia. When the changes went live, we had to spend numerous hours fixes all the bugs in our own framework caused by the Celestia update.

Go-Gitters Deliverable 5: Celestia

Fault Injection Overview:

We have an a single include file with multiple functions to make testing easier. The changes we made to the code are as follows:

1. Changed the circleArea function by removing the parenthesis surrounding r * r. This causes the code to return a number when a negative value is passed as the radius.

2. Changed the sphereArea function by removing the parenthesis surrounding r * r. This causes the code to return a number when a negative value is passed as the radius.

3. Changed the pfmod function to check for a zero value in the denominator of the input. This causes zero to be returned instead of nan.

4. Changed the clamp function to return any value under 1 as 0 and any value over 1 as 1. No longer returns a range and instead only produces 0 or 1.

5. Changed the sign function to return the inputted value when it is negative, instead of returning 1.

Report:

We had our easiest time of the entire project on this deliverable. Most of the code that we used is math based and is relatively easy to inject faults into a math function without ruining the entire test framework. There was something oddly satisfying about adding failure points to the code we spent so long trying make perfect.

Once again, time and planning were the most important barriers for this part of the project. Without an adequate test plan we mostly just stumbled through the creation of the fault cases. In our case, it was fortunate we chose a project that is built using a large portion of math. Had we been unable to understand more portions of the code we used then creating fault injections would have been considerably more difficult.

Final Report

Overall, we feel that this project was a major challenge. We did not have much direction after choosing our original open-source project, and with no firm timetables or check-in points to receive criticism or assisstance, we mostly just floated along until we had to have something for a presentation. Additionally, creating a framework for Celestia was a much larger and more difficult task than we initially thought. It is a huge codebase and has a lot of very good developers working on it. It is also written in C++, which none of us knew, and learning a new language as well as trying to navigate the new codebase was daunting.

As we have stated numerous times, our teamwork could have been better from the beginning. No one really took charge and became a leader and so we weren't held accountable at all for anything we did. This project was more like putting out forest fires when they started rather than a controlled burn of the underbrush. We learned the value of creating our own schedule to follow and to spend more time planning and less time implementing. It was also

really great to work with new technologies and spend some time in a real open-source project to get a feel for how a real code base is used.

Team Evaluation:

Overall our team performed well when we knew what needed to be done. The only complaints any of us had was the amount of other work we had and lack of time we had to devote to Celestia. Our planning wasn't the best, but as the semester progressed we got better about picking up the individual tasks we had to complete. As we have stated before, the scope of the project was larger than we initially believed and we felt a little overwhelmed for a large portion of the semester. In the end we are extremely happy with the way our framework turned out.

Project Evaluation:

We enjoyed working with an open-source project but having to choose one instead of being assigned one felt a little off. Some groups managed to pick projects with loads of pre-built test cases and clear direction for how to work with the code, but other projects barely had readable source code. Also, maybe in the future, there could be designated times for teams to receive feedback from the professor directly. The pacing of the project deliverables felt fair and having the entire semester to work on the project is extremely beneficial.