Robots and Robots and More

Alex Siegel

KENNEDY SPACE CENTER

Major: Computer Science & General/Electrical Engineering

Program: NIFS Year-Long Intern

Mentor: Caylyne Shelton

Date: 07/03/2018

Designing an Automated Testing Framework, Updating "Swarmies", and Performing Rendering Artifact Detection

Author: Alex Siegel

Swarthmore College, 500 College Ave., Swarthmore, PA, 19081

Nomenclature (in order of appearance)

GUI = Graphic User Interface

CSCI = Computer Software Configuration Item (team)

LCC = Launch Control Center

COTS = Commercial-off-the-shelf

SSD = Solid State Drive

QOL = Quality-of-Life

VAB = Vehicle Assembly Building

QR = Quick Response Code

SSH = Secure Shell

I. Introduction

Over the course of this year I worked on several projects across different departments, from physical robots at Swamp Works, to automated robots running functional tests on launch critical Graphical User Interfaces (GUIs). My overarching project this year was designing a new automated testing framework. It began with becoming familiar with the existing automated testing framework, which used image matching to perform programmatic GUI testing. After writing a few dozen tests with the existing framework, I was able to identify the strengths and weaknesses of the system and begin the search for another tool to replace it. I ended up selecting an open source library, which I modified to fit both National Aeronautics and Space Administration (NASA) requirements and those of the specific Computer Software Configuration Item (CSCI) I worked on. After the new framework was feature-rich enough to meet the team requirements, I helped roll out the changes, managed the transition to the new framework, and directed continuing automation work in regards to the library. The result was a comprehensive new way of efficiently testing the GUI that was robust enough to handle small changes to the GUI itself.

Additionally, for approximately six weeks I worked on rebuilding, repairing, and updating the software of four "Swarmie" robots for a NASA event. By the close of the six weeks, the four robots were operational and had improved obstacle detection software that solved a faulty stop signal issue that the robots were experiencing.

The third major project I worked on was an automated detector that ran in one of the Firing Rooms in the Launch Control Center (LCC). The detector attempted to recreate a rare and critical bug where launch data was failing to update when being obscured by another window, resulting in half-updated rendering artifacts appearing on the screen. I ended up writing and running three different versions of the artifact detector throughout my internship, and though they detected no valid artifacts, they helped rule out many possible causes of the rendering error.

II. Projects

A. Automated Testing Framework

When I began working at NASA, the current automated testing framework was a merger between two open-source pieces of software—a programming framework and an image matching library. The image matching tool was used to detect parts of the GUI with which to interact on screen; the programming framework and language governed interaction with the GUI. After a week spent learning and familiarizing myself with the software, I began to write my own automated tests using the tools. I wrote approximately fifty tests, helping to automate my CSCI's test procedures, before I saw both the strengths and weaknesses of the tool. The tests were written in a high-level programming language close to English, which made it advantageous for this type of testing because of its readability and ease-of-use—especially for the non-developers who were expected to write many of the tests. The image matching library was a useful tool because it could detect colors, text, and images onscreen, but it was especially prone to being broken by even small changes to the GUI. Enabling anti-aliasing on text, changing fonts, and adding small environmental differences between two machines would be enough to break every image matching test. These errors would need to be individually debugged and patched (often resulting in the

laborious work of retaking hundreds of new images). It was a time-consuming, tedious task that required a developer with a lot of specialized experience with the library itself.

After several weeks of work on creating more automated tests, I was told that a new type of middle-ground test was expected to be created, called a Combined Functional Test (CFT). It was intended to bridge the gap between the broader system-level tests and the narrower focus of the level 5 tests, which validated basic requirements (i.e. sending an invalid command and verifying it would fail as expected). Since CFTs were soon to become commonplace, but were still in the planning phase, there was an opportunity to find a new framework to use to write them. A new framework offered the potential to avoid the disadvantages of the previous version, in addition to providing greater functionality. I was tasked with searching out good alternatives in both COTS and open-source software to find a suitable replacement. In general, the COTS software was reliable, easy-to-use, and easy to set up. Unfortunately, it was also bloated, brittle, and expensive. For example, the main COTS product I looked at had the useful functionality of being able to turn on a "recording" mode, step through a test by hand, and then replay the test on its own. The downside, besides the cost, was that the test it produced was a huge file, that couldn't be modified programmatically. If part of the test needed to be changed, the whole pre-recorded procedure would need to be redone.

I ended up selecting an open-source solution. It was free, feature-rich, and compatible with the existing test automation framework. Instead of using image matching, this library was able to hook into the Java hierarchy of the GUI and interact with the components themselves based on the hidden component names. This meant it was resilient to changes in the GUI, because it was unlikely that the components themselves would ever be removed or renamed. I wrote several demo scripts with the library, and made a pitch to my technical lead, Jason Kapusta, to accept the library as the new tool for writing CFTs. After it was accepted, I began to work on modifying the library to be suitable for CFT work. The main obstacle was the GUI on which we were performing the test. Most of the code behind the GUI was launch-critical code and could not be easily changed, but several of its features made it incompatible with the library. The three most critical missing features from the library were the ability to hook into the main JWindow at the top of the GUI, find modal dialogs, and use popup menus (as well as adding names to all the components so they

would be discoverable by the library). I added a new Java function operator and corresponding operator factory to the library that gave it the functionality to select JWindow components, like the one at the top of the GUI. I modified another function to allow it to select our left click enabled popup menus. Lastly, I modified the library in order to allow selection of modal dialogs by expanding the scope of context in which the library would search for a matching dialog.

After I finished making the necessary patches to the library, I was able to roll out CFT work. I made several example programs and distributed them to other interns, the automated test team, and my CSCI. I spent the remainder of the time patching issues people encountered with the library, helping people get the library working for them, and expanding its functionality.

B. "Swarmies"

[2] "Swarmies" in front of the VAB at KSC

For a period of approximately six weeks I was tasked with repairing several "Swarmie" robots in the Swamp Works lab at Kennedy Space Center. The "Swarmies" are Roomba-like remote controlled robots with an attached claw for picking up objects. They also come equipped with several cameras and distance sensors. They have a toggleable automated mode, during which they wander an area looking for QR-code stamped blocks to gather. There were four robots that needed to be repaired for a NASA competition. These robots were in various states of disrepair. Two were fully disassembled, one was experiencing a corrupted software issue, and another had damaged

hardware (a snapped claw and support struts as well as non-functional servos). After a tutorial on how to load Ubuntu Linux onto the robots, build their software, and put them together, I set to work repairing the robots.

The hardware fixes were straightforward, but the software differences between the robots resulted in a prolonged debugging process. I was eventually forced to wipe each of the robots clean and reload all their software. One of the robot's controllers was faulty, but after thorough investigation, I came to the conclusion that the wireless chip in the controller itself was broken which meant it was essentially irreparable. At the end of this first few weeks, all four robots were in a state of good repair, but missing the one controller. To ensure the robots were working correctly, I tested the robots for several hours a day, having them run their automated movement course, as well as manually controlling them to do what would be common tasks for the upcoming competition. "Swarmie" operators had previously experienced an issue where the onboard Intel NUC computers would fail to get past the boot screen when powered up. After a few days of testing, I encountered a recurrence of this issue. It turned out to be a bad SSD, but since there were no replacement SSDs on hand I searched for a workaround. Pulling the power from the onboard battery to the NUC, leaving it unplugged for ten seconds, then plugging it in and restarting the device, temporarily got the robot back on its feet and allowed it to boot properly. It was only a temporary workaround, but resetting the devices in this way abated the problem for roughly a day, so I performed the reset on all four robots immediately prior to the start of the competition and all performed nominally. Afterwards, I replaced the SSDs on the broken robots which fixed the issue permanently.

One Quality-of-Life (QOL) problem I noticed during the manual testing phase was that when the robots attempted to pick up certain blocks at an odd angle (or stacked blocks), their claw would sometimes raise the blocks to a position in front of the distance sensors mounted on their face. This would trigger the obstacle collision code that would lock the robot's wheels in place, preventing it from moving forward. The workaround was simple—either turn the robot, move backwards, or lower the claw, and it would be able to continue on its way. Although this was uncommon, happening an estimated once in every thirty block pickups, the fix was unintuitive for first time users (as all the participants of the competition would be). I wasn't at liberty to remove

the obstacle detection code because the nature of the competition made the robot vulnerable to being smashed against a wall. Therefore, I modified the collision code so it had a minimum distance (from which a wall would not be detected), slightly past the length of its claw. This meant that most blocks it picked up were inside the "safe zone" for the collision detection code and would not cause the wheels to lock. However, even after this change, some blocks were barely slipping out of the minimum distance I had set, and causing the robot to stop. I was wary about expanding the minimum distance by much because of the scenario in which the robot would rotate into facing an immediate wall, only inches away, and not detect it as an obstacle. I added another check to the obstacle detection which checked whether all three distance sensors were triggered, or only one. If it was just the middle sensor, I instructed the robot to ignore the collision warning. The idea behind this change was that walls usually triggered all three distance sensors, whereas only thin pillars might cause a single distance sensor to ping. More importantly, the blocks that the claw raised up in front of the distance sensors triggered only the middle sensor for the most part (since they were not particularly wide objects). This change was not sufficient on its own, since on rare occasions a block could spill over or be picked up at an odd angle and still trigger multiple sensors, so it had to be used in conjunction with the minimum distance change to eliminate all chances of incorrect collision detection. I also added a greater minimum distance specifically for the middle distance sensor so that its output would still be useful. If it detected a thin pillar at a distance, it could still trigger the obstacle detection code. The only error case was a thin pillar that the robot turned into, so it entered into the middle sensor "safe zone". However, this scenario was rare enough that my mentor was willing to take the risk for the QOL benefits of the change.

C. Artifact Detection

The first project I worked on when I became familiar with the automated testing framework and associated image matching library was an automated artifact detector that ran on development machines and workstations in one of the Firing Rooms in the LCC. There was a rare bug that was being encountered based on an interaction with the Launch Control System GUI and other frames superimposed on top of the GUI. The result was that data being updated on the GUI's displays would only partially update—data obscured by another frame (whether a terminal or another program) would fail to change, but visible data would update correctly. If the window was moved away, artifacts were left on-screen—half-updated and half-old information. These artifacts would

usually fix themselves if you moved the mouse over them, but the problem was still critical because the users needed to be able to depend on seeing accurate data from the GUI without needing to worry about double-checking to make sure the data had updated correctly. We weren't sure what was causing the issue, but it was essential to be able to replicate the bug to determine exactly what conditions allowed it to occur. While a workaround was possible, it involved a refresh of everything on-screen every few seconds, so it was extraordinarily performance-heavy.

For the first iteration of this artifact detector, I was instructed to mimic the Launch Control System GUI with a Java program that was a large JTable that took user input and then filled all its columns and rows with that input. It simulated updating "data" by receiving different input from the testing framework, displaying that input in every cell of the JTable, and verifying it had correctly updated its cells. My robot program launched the Java program, verified it opened correctly, then entered different user input every few seconds and verified that the JTable changed as expected. If the result differed from the expected result, the program took a screenshot and filed it away, then continued running (for weeks if uninterrupted). After several weeks' worth of runtime, the program caught nothing. Even if it had encountered a genuine artifact, it was unlikely that it could have successfully identified it because the accuracy parameter it was using was too low to notice a few cut-off characters. Furthermore, it was missing several key elements that lead to the error occurring in the first place.

For the second version of the artifact detector, I was instructed to include many of these missing elements, such as a "blocker" window that popped up, obscured the data table, and then disappeared again. I also raised the accuracy parameter, and wrote failure-case tests to verify that artifact-like objects would be detected and logged. The other changes I made included bursts of activity then rest periods (similar to how a user interacted with the actual GUI), changing background colors for the table (similar to the flashing alarms of the GUI), and minimizing, then refocusing, the table itself. I checked before and after each of these events to verify that the table didn't show any signs of artifacts. The last alteration I made was to have two instances of the program running simultaneously on the same computer's monitors, but both SSH'd into different computers. This was to simulate how we run the actual GUI, with different processes running over different servers. As of the writing of this paper, this version has not caught any artifacts, but it is

more promising because it has produced far fewer false positives, which increases the likelihood that it will detect a valid artifact if it encounters one. If this version produces no artifacts, then the scope of the problem is narrowed further to an interaction with the actual GUI and the frames that obscure it. Therefore, the next version that I will work on will run a genuine display from the GUI, instead of a JTable that simulates a display, and verify the same conditions.

III. Conclusion

While I still continue to work on the rollout of the automated testing framework and future iterations of the artifact detector, the most technically difficult parts of the assignments are behind me and I look forward to seeing the results. The "Swarmies" were a resounding success at their competition and worked as expected (bar one or two controller glitches). Now, they sit in Swampworks, repaired, ready-to-go, and equipped with upgrades to their software.

The artifact detectors have yet to find valid artifacts, but their existence and their various iterations help narrow down the possible causes of the rendering issue. Eventually, one of them or their kind will be capable of replicating the error on demand and the bug will be able to be identified.

My overarching project, the automated testing framework replacement, is patched, ready-to-use, and can successfully accomplish all the requirements of the framework it was designed to replace—and more. Within the next few weeks, and continuing on, people will be writing CFTs and other pieces of software to support the library, and soon it will supplant most of the existing automated testing effort. It should have a long future as a piece of software because of its resilience to GUI changes, ease-of-use, and consistency with its results. There are some applications where the old image matching framework will still be necessary, but hopefully this new library is the first step on the path towards the reduction of manual human-run testing and arduous image-matching debugging. Between the physical robots and the virtual, this internship has been especially rewarding and I believe the work I have accomplished will prove useful for my CSCI and the greater Software group at Kennedy Space Center.

Acknowledgments

Thank you to Sam Goff—I couldn't have done it without you. Thank you to Will Denis, and Corey Dike for your willingness to help, your advice, and your camaraderie. Thank you to Jamie Szafran, for your mentorship. Thank you also to Jason Kapusta, Jordan Kiser, Tony Ciavarella, Caylyne Shelton, Jill Giles, and all those who have offered me help along the way.

References

[1] Swarthmore Photo
https://www.swarthmore.edu/communications-office/college-logo

[2] "Swarmies" Photo
https://www.nasa.gov/content/swarmies-shuffle-through-field-tests