Autograding Interactive Computer Graphics Applications

Evan Maicus
Rensselaer Polytechnic Institute
maicue@rpi.edu

Matthew Peveler
Rensselaer Polytechnic Institute
pevelm@rpi.edu

Andrew Aikens
Rensselaer Polytechnic Institute
aikena@rpi.edu

Barbara Cutler
Rensselaer Polytechnic Institute
cutler@cs.rpi.edu

ABSTRACT

We present a system for the automated testing and grading of computer graphics applications. Our system runs, provides input to, and captures image and video output from graphical programming assignments. Instructors use a simple set of commands to script automated keyboard and mouse interactions with student programs at fixed times during execution. The resultant output — including plaintext standard output and mid-execution screenshots and `GIFs` — are displayed to the student to aid in debugging and ensure compliance with assignment specifications. Student output is automatically evaluated by basic text and image difference operations, or via an instructor-written validation method.

We evaluate the success, implementation, and robustness of our design through deployment of this work in our university's senior undergraduate/graduate computer graphics course. In this course, students implement a variety of graphical assignments using `OpenGL` in `C++`. We summarize student feedback about the system gathered from anonymous end-of-term course evaluations. We provide anecdotal and quantitative evidence that the system improves student experience and learning by clarifying instructor expectations, building student confidence, and improving the consistency and efficiency of manual grading.

This research has been implemented as an extension to Submitty, an open source, language-agnostic course management platform which allows automated testing and automated grading of student programming assignments. Submitty supports all levels of courses, from introductory to advanced special topics, and includes features for manual grading by TAs, version control, team submission, discussion forums, and plagiarism detection.

CCS CONCEPTS

• **Computing Education** → Computer Science Education; • **Computing Methodologies** → *Computer graphics.*

KEYWORDS

Computer Graphics, Testing, Autograding, Course Management

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6793-6/20/03…$15.00
https://doi.org/10.1145/3328778.3366954

ACM Reference Format:
Evan Maicus, Matthew Peveler, Andrew Aikens, and Barbara Cutler. 2020. Autograding Interactive Computer Graphics Applications. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), March 11–14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328778.3366954

1 INTRODUCTION

Due in large part to the recent explosion in computer science enrollments, extensive work has been done to develop automated grading solutions to aid in instructor course management [2]. While prior research has been done regarding the importance of these systems [17], that work tends to heavily focus on the impact of autograding on introductory courses [16], and in particular on non-visual assignments which take simple, text-based input and which produce text-based output. As introductory course sizes have swollen, however, advanced-topics course enrollments have grown as well. This has left a need for autograding solutions for assignments which do not fit into the traditional, text-based grading scheme. Such is the case for computer graphics assignments.

1.1 Manually Grading Graphics Assignments

The manual evaluation of student graphics assignments is a tedious and time-consuming task. For compute-heavy simulations, such as ray tracing, minutes of execution may be required before a gradeable visualization is rendered. If an assignment requires manual interaction — perhaps via keyboard or mouse input — the grader must stay nearby, ready to manually manipulate the assignment to keep the test-suite moving. Assignment grades are often determined by manually manipulating a program to inspect for visual errors. This visual output is more difficult to capture than text-based command line output. As course sizes swell, the amount of instructor and TA time spent on these monotonous tasks grows, decreasing time available to teach and help students in office hours.

Furthermore, student programs that interface with low level graphics APIs such as `OpenGL` [8] may have dependencies on specific hardware, drivers, or operating systems. These programs may execute differently from system to system, yielding different output or code failure [10]. When the student and grader use different environments, efficient and fair evaluation of student work can be negatively impacted. When a program fails on a grader's machine, it may be unclear whether the student work is incomplete and would fail on every system or if perhaps the error is environment-specific, and should receive only a minor penalty or no penalty at all. The grader has three unsatisfying options: spend additional time attempting to debug the error, delay grading and contact the student for followup, or give the student no credit.

1.2 Contributions

We present the design and implementation for the automatic execution and evaluation of student programs with graphical front end. This work is integrated into Submitty, an open source, full-featured assignment submission, autograding, and course management environment [14]. This system was deployed in RPI's Advanced Computer Graphics course using `OpenGL` and C++. Our system:

- Allows the instructor to script keyboard and mouse input actions that will be applied to a running student process.
- Provides responsive feedback to students, including text-based output, screenshots, and videos of execution.
- Facilitates visual image difference operations for automated evaluation and scoring. The instructor may specify custom validation methods to appropriately judge the correctness of nuanced assignments and award full or partial credit.
- Optionally generates output files from an instructor-prepared solution, against which student work will be compared.

2 RELATED WORK

Few existing systems are explicitly designed for automated grading of graphical applications; however, various methodologies are employed in industry to test and validate graphical software products. We pay special mind to solutions which support `OpenGL`, the graphics interface used in our university's computer graphics course.

2.1 Defining Full and Partial Correctness

To automatically grade or validate visual output, an appropriate definition of "correctness" must first be established. Under one definition, correctness could be the full-color RGB, pixel perfect match of an output image. An alternative definition may be the perceptual equivalence to a target visual output [6]. Then a comparison algorithm such as Mean Squared Error (`MSE`) — which relies heavily on image color — or Structural Similarity Index (`SSIM`) — which relies on the relative structure of an image — can be chosen to evaluate an image. Automated visual evaluation of graphical program output provides a close analog to the method of manual grading described in section 1.1. Yet it is important to remember that differences in hardware, drivers, and numerical implementation of a graphics program can lead to outputs that differ slightly from machine to machine [10, 18].

Not all schemes for determining graphical application correctness utilize a program's visual output. The correctness of a graphics application may instead be determined by evaluation of text-based output or internal program state. For instance, an automated grader might test the value of the OpenGL `GL_MODEL_VIEW` matrix when teaching transformations. The grader might unit-test individual student-written functions and inspect their return values rather than provide an end-to-end, holistic assessment of visual program output [18]. An automated grader of 3D meshes could validate the expected number of each type of polygon and the positions of vertices in the mesh [5] or use a method of geometric approximation [3]. It is important to include an appropriate tolerance or epsilon in all comparisons – numeric and visual.

Techniques that are appropriate for regression testing of software products may not be directly applicable to grading student work. To avoid demoralizing students, the grading of their work should generally not be binary. Instead it should evaluate partial correctness, and highlight incremental progress and improvement in the quality of results over multiple submissions.

2.2 Automated Interaction

There are a number of ways to provide automated input to running graphical applications. If the user interface is highly structured (e.g., interfaces written in markup languages such as `HTML`), we can leverage language-based element selectors for detection and interaction [12]. Such test infrastructures are especially useful when the components of a GUI may not always appear in the same pixel coordinates. If an application has a highly predictable GUI, but is not written in a structured language (e.g. a GUI programmed in `OpenGL`), Optical Character Recognition (OCR) and image recognition algorithms can be used to detect the existence and position of GUI elements [15]. In cases in which a structured language is not used and GUI features are not easily detectable or are non-existent, inputs can be either recorded or hand specified [4].

While industrial applications exist for testing computer graphics applications, and specifically `OpenGL` based graphics programs, most are closed-source, require payment or subscription to use, and are not integrated into an automated grading environment. During our literature review we did not encounter any automated grading systems, for pay or otherwise, which natively facilitated keyboard or mouse interactions with student programs.

2.3 Testing Methodologies

Testing graphical programs for correctness has been thoroughly investigated within the television [6], web design [12], and video game [1, 10, 15] communities. To ensure a software product works in all target environment deployments, a suite of unit-tests can be deployed across many different hardware/software configurations [10, 11]. While thorough, such an approach can be prohibitively costly in time and hardware, especially for small projects.

Experimental work has been done to test graphical applications by deploying roaming, error-detecting artificial intelligence agents [7] and define classes of visual error with algorithms for their detection. Alternatively, some frameworks deploy artificial users that have been trained to properly interact with an application at a far faster rate than a human user. Event logs of these interactions are then stored and later evaluated for error messages or crashes [1]. Such designs require additional up-front investment from the test-writer, who must either accurately identify a representative set of bugs and detection algorithms, or program relatively realistic artificial users. However, these designs remove the need to configure dozens or hundreds of unit-tests. Deep learning has been used to train agents to evaluate and classify the correctness of randomly-synthesized variants of graphics applications [19].

3 SYSTEM DESIGN AND IMPLEMENTATION

Our system was designed to support RPI's Advanced Computer Graphics course, which is comprised of five intensive, multi-week OpenGL and C++ programming assignments. The instructor provides a common framework of initial code for each assignment, and a structured set of requirements to complete core features of an algorithm and optional extensions. We have designed most of

our testing to rely on simple per-pixel image differences, with an adjustable tolerance based on the assignment topic. We anticipate exploring the use of a perceptual correctness metric [6] and more complex computer vision algorithms for analysis in future work.

We use end-to-end test suites as our primary testing methodology rather than short unit tests of student functions. We facilitate instructor scripted keyboard and mouse input to running student applications. This approach is language- and application- agnostic, allowing our system to interface with non-OpenGL assignments.

Our system was designed for integration within Submitty, the Open Source automated grading and course management tool used at RPI. Students are able to submit their assignments via Submitty's intuitive web interface. Our extension then handles submission processing before reporting results to the student via the same interface in a timely manner.

3.1 Window Detection

To receive keyboard and mouse input, we must give the student application window focus and calculate its relative position. Three methods were considered to accomplish this:

- **Window Title** The instructor could specify a mandatory window title for each assignment, which can be detected by inspecting the metadata for all running windows. However, if a student does not follow the window title specification, the resulting failure for visual output to be detected by our system might be difficult for them to debug.
- **Process Id Ownership** Most applications include the owner process id in the window metadata. Using `Unix` utilities, we can detect window ids that belong to a student process. While detecting window ids is effective for detecting the windows of `OpenGL` programs, windows generated by the standard `Python` graphics library, `Tkinter`, for example, do not store initializing process ids.
- **Newly Created Windows** Before launching the student process, we record a list of all open windows on the system. Then, after the student process begins, the first new window that appears can be assumed to belong to the student process. While simple, this method is less robust and may errantly capture unrelated popup windows if they open during the critical moments of application launch.

We have selected a flexible hybrid approach to detect window ownership. For the first second of student application execution, we search for a graphical window by process id. If that method is not fruitful, we switch to the newly created window approach.

3.2 Input Specification

Interactions with student assignments are performed using window pixel coordinates. This decision facilitates interaction with arbitrary student assignments, regardless of programming language or software structure, and fits the needs of our targeted course. This design choice can be extended in the future to use element-detection, `OCR`, and image recognition solutions [13, 15].

Our comprehensive list of supported actions is as follows:

(1) **`delay:`** A wait time, in seconds, before the next action.
(2) **`type:`** A string typed in window as keyboard input. Multiple key combinations such as `ctrl+alt+del` may be specified.
(3) **`stdin:`** A string delivered to the process as standard input.
(4) **`stop`**: Terminate the student process.
(5) **`start`**: Re-invoke a stopped student process.
(6) **`screenshot:`** Save the current window to a file.
(7) **`GIF:`** Capture a video of the current window to a file. Capture and replay rates specified in frames per second.
(8) **`mouse move:`** To a specific window coordinate.
(9) **`click:`** Press (or hold) left/right/middle mouse button.
(10) **`click and drag:`** Move mouse while holding mouse button.

For each testcase in an assignment, the instructor prepares a sequence of the above actions. Most actions, such as keyboard inputs and mouse clicks, are processed nearly instantly. Compound actions such as `click and drag` can take more time to perform. Timing can be further padded with delays, e.g., to allow a ray tracing application to completely render before taking a screenshot. Most actions include a `repeat` option after a short delay, allowing easy specification of "double click", for example.

3.3 Fault Tolerance

Automated grading is different from industrial software product testing in a number of significant ways. Product tests usually take the form of "Regression Tests", which we assume passed in the master branch, and we simply want to confirm the proposed software changes to not break this functionality. On test failure, finding the source of errors is typically a search process focusing on the modified lines of source code. In automated grading, we are testing a new software system that instead aspires to be able to succeed against a battery of tests. For this reason, it is essential that we capture evidence of partial work, not only to fairly reward student effort, but also so that we can deliver all available information about their run to facilitate debugging.

A student application might crash at any time, including during the processing of a keyboard or mouse action. In fact, actions triggering computation in student code are likely the source of most crashes. For this reason, once the program window is detected, its status is polled at the start of each action, and between sub-actions for compound commands such as `Click and Drag`. If a window close occurs during an action, the action immediately stops, and no more actions are delivered until another valid window appears. Note that we handle applications which spawn and close windows, so long as multiple windows are not created concurrently. If a student program crashes, all text output, screenshots, and `GIFs` recorded up to that point are delivered to the student, as well as any additional `stderr` output dumped by their program or `OpenGL`.

3.4 Security

Autograding differs from industrial software testing in that student code is executed as a black box. Student programs must not be allowed to affect the greater testing infrastructure. We must ensure student privacy and fair and equal access to testing environment resources. We must guard against the small but real chance that student code may be malicious.

Submissions processed by Submitty are run within a *Jailed Sandbox*: a carefully permissioned environment in which an untrusted user executes an application and in which system resource usage is

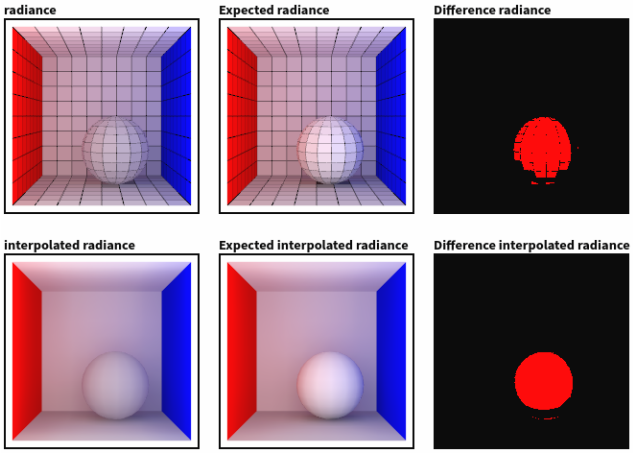

Figure 1: A mathematical error in the surface area term of radiosity will produce subtle variations in lighting. A side-by-side, per pixel image comparison aids student debugging.

carefully monitored. Alternatively, we could use containerized autograding to ensure security and resource management [9]; however, at the time of our development, we found that container solutions did not natively support running graphics applications.

To run applications which interface with a screen, the restrictions of a jailed sandbox had to be carefully loosened. At the beginning of the execution of a graphics application in Submitty, an untrusted user is granted access to the `OpenGL` distribution installed on a grading machine, and necessary system variables are set (e.g., "`DISPLAY`"). The user is then given access to the running `xserver`.

Another security concern is that keyboard and mouse actions might errantly affect the test system. Thus, while processing a suite of actions, it is important that the mouse does not escape the bounds of the application window created by a student program. To ensure this, we store the dimensions of the student window, and any autograder actions which attempt to move the mouse beyond these bounds are either clamped to one pixel inside of the window's edge (to avoid window resizing), or, in the case of actions such as `Click and Drag`, are decomposed into multiple, smaller actions.

3.5 Validation

To facilitate side-by-side manual comparison or automated grading, we must have solution images against which student output will be compared. When preparing multiple complex test cases which utilize lengthy sequences of keyboard and mouse actions, manual preparation of solution images is time-consuming. As an alternative, we allow instructors to provide their assignment solution, which is used to generate text output, screenshots, and video for use as ground truth values.

Our system compares screenshots using a simple image difference operation with a customizable pixel percentage tolerance. By default, we compare images using the `MSE` operation [6], which grades students more heavily on color than on structure. However, for specific assignment tasks, other metrics may be more suitable. Instructors can write custom validators that return a score and optional text and/or image feedback for the student. These validators

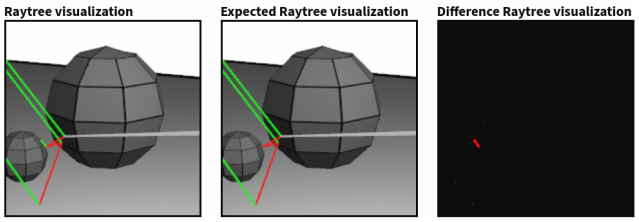

Figure 2: A single pixel ray trace tree visualization helps students spot a missing shadow bounce and other geometric errors in their recursive ray intersections and reflections.

may leverage image processing libraries such as `OpenCV` to perform in-depth analysis of student output.

4 RESULTS: CASE STUDY

In Spring 2019, our system was deployed and extensively used in RPI's Advanced Computer Graphics course to test and evaluate five multi-week programming assignments using `OpenGL` in `C++`. The assignments cover a variety of topics including: mesh processing, cloth and fluid simulation, radiosity, ray tracing, photon mapping, shadow volumes, and procedural shaders. All of these assignments were run successfully using our system, and the captured screenshots and videos were presented to students shortly after each submission. These visual results from each test case were also autograded to varying degrees of success (some assignments and topics were more conducive to effective automated scoring). At the end of the term, students completed a course evaluation which included questions regarding their experience using our system.

4.1 Grading Time-Intensive Computation

The test suite for a radiosity + ray tracing + photon mapping assignment was the longest-running (cumulative time for all actions including delays) of all five assignments. The action sequence include delays as long as 60 seconds before an intermediate screenshot was saved, and a total of twelve and a half minutes of explicit delay was specified over all testcases. The manual grading time savings of our system was more evident in this assignment than any other, as the grader did not have to monitor the renderings during testing, but could instead quickly review saved imagery and correct automated scoring if needed. Prior offerings of this course used the same assignment but did not use automated testing. It is unlikely that the grader was equivalently patient or thorough in manually running student code. We presume student code was terminated earlier and evaluation was potentially inaccurate because it was based on partial test results. The `MSE` image difference was effective in validating rendering algorithms and generally useful in debugging and awarding partial credit (Fig. 1 & 2).

4.2 Step-by-Step Visual Unit Testing

When testing student implementation of shadow volumes, it was helpful to decompose a longer sequence of test actions into a series of intermediate screenshots with models of different geometric complexity, different visual debugging options, and different light positions and camera angles (Fig. 3). The presented sequence of

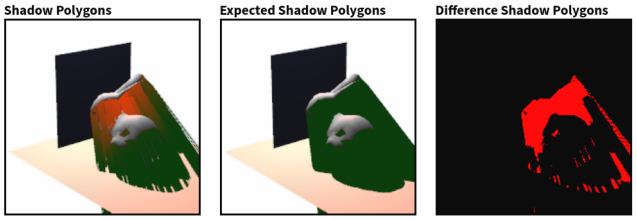

Figure 3: In addition to capturing the final stencil buffer shadow rendering, we also visualize the shadow volume polygons for different camera and light positions – enabling students to spot geometric inconsistencies in their code.

output helped students to determine what worked vs. what was buggy, and to hypothesize and diagnose their errors. Preparing these tests was made easier by years of instructor familiarity with common student implementation errors, which is testament to the fact that writing good autograding testcases requires experience.

4.3 Wireframe for Grading Mesh Operations

The students were asked to simplify a 3D triangular mesh by editing and removing elements in a greedy manner. Using a simple `MSE` comparison of the mesh can confirm that the overall shape and shading is approximately preserved, but it does not necessarily reveal whether the student implemented a specified deterministic algorithm for element reduction. By overlaying a wireframe on the mesh, we were able to fully test student output (Fig. 4). However, while this did allow us to score student output, it is not simple for the student to use the produced difference image to debug their assignment. Providing better feedback likely requires more thought for assignment-specific test case design and is left as future work.

4.4 Physics Simulation Videos/Manual Grading

On another assignment, students implemented an explicit Euler integration mass-spring physical cloth simulation. Their simulations were initialized with forces and a reasonable timestep and allowed to run for approximately double the expected time necessary to reach equilibrium. Through the automated grading of this assignment, the instructor found a subtle, long-standing error in the assignment specifications for damping coefficient, that yielded a visual mismatch of the equilibrium state between correct student solutions and the incorrect instructor solution. Student grades were manually inspected by the TA and appropriate adjustments were made to the scores of impacted students. In this instance,

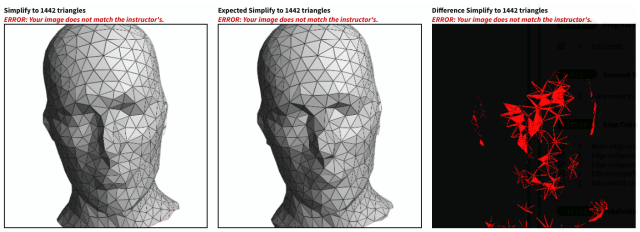

Figure 4: We were able to detect errors in geometric simplification by rendering the mesh wireframe.

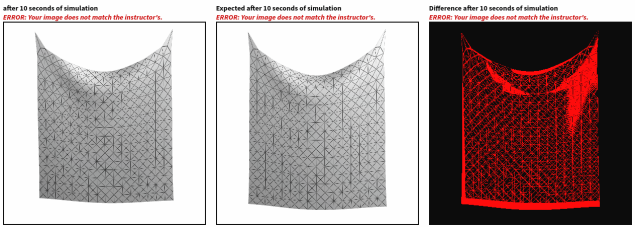

Figure 5: It can be difficult to discern differences in assignment performance and simulation timestep from a single image. In this case, a `GIF` revealed that the student's simulation ran too slowly for our 10 second test-suite, but did reach an acceptable point of equilibrium.

our system's ability to capture and replay `GIF` videos of student code execution was invaluable, as differences in performance and simulation timestep cannot be discerned in a single frame (Fig. 5).

4.5 Creative Solutions for Nondeterminism

To introduce rotation, scale, and translation transformations, students are asked implement a simple fractal rendering algorithm that uses random points. We found that a naive image difference using a reasonable but small number of points yielded poor scores for correct solutions or require such a large tolerance that it might fail to identify buggy student solutions. To mitigate this problem, we reduce the size of the screen capture and increased the size of each rendered point (already options required to be implemented in the assignment specifications). The simple image difference only required a small tolerance factor and correctly distinguish correct and incorrect implementations (Fig. 6).

4.6 Automated Execution Without Validation

To familiarize students with the *GPU* rendering pipeline, they were asked to implement a few simple fragment and vertex shaders (e.g., a *brick* shader or a *wood* shader). Because the instructions were open-ended and encouraged creativity, automated validation was not used. However, the system was still configured to run student submissions and collect screeenshots for later manual grading of subjective quality – saving grader time and hassle! As an added benefit, when students submitted, they were able to confirm their program was working correctly by inspecting these screeenshots.

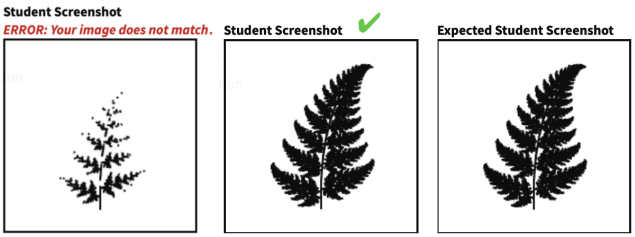

Figure 6: An error with probabilities produces a buggy image (left). Debugged student work (middle) is awarded full credit despite not being a per-pixel match with the expected (non-deterministic) output.

4.7 Course Evaluation Responses

Students completed an end-of term course evaluation about their experience submitting graphical assignments for automated testing. 27 of the 34 students enrolled in the class returned the survey and most provided lengthy, thoughtful written responses to our questions. Of the 27 respondents, 19 found our extension helpful to their learning in the course, 6 did not specify, and 2 students found our system unhelpful. Comments included that "seeing side-by-side comparisons of the student and instructor solutions was very helpful". Of the respondents, 4 commented on long wait times while using our system. One student stated that their "biggest gripe was the slow speed of the grading," and another that "the time it takes to autograde is a bit frustrating, although it does incentivize early submission." 18 of our 27 students noted that the tolerances used in our image difference metrics did not afford sufficient partial credit, and, in some cases, resulted in false-negatives – cases in which the student felt that their output was subjectively equivalent to the instructor output, but in which the autograder's tolerance awarded them partial or no credit. Students noted that "sometimes it (the autograder) would give deductions even when differences were minimal enough", and that "the autograde 'grade' is really the only thing that was kind of stressing." Students unanimously appreciated our system's ability to show them a comparison of their output to the correct, instructor output, noting that this comparison helped them to detect errors early, and therefore provided opportunity to make adjustments earlier in the development cycle. Feedback included that "diff view was helpful for more subtle issues" and that "gifs and images with the differences given are really helpful".

5 DISCUSSION

Most students found our system to be helpful to their learning – in particular, its ability to show them the expected output. Throughout the semester, we were pleased with the system's performance across a variety of assignments, and with each homework we found new ways to leverage its functionality. Furthermore, we noted that while using our system students made earlier test submissions, which allowed them to confirm through automated testing and grading that they were on the right track and it motivated them to tackle the next challenge. As a result, the average student successfully completed more of the milestones per assignment than in past semesters. This makes sense, since previously students did not receive any feedback on their submission until manual grading was completed, often many days after the homework deadline. An additional benefit of running student assignments through a set of automated tests was that it allowed the instructor to specify more complete and rigorous testing than would otherwise have been performed, especially when grading long-running assignments.

Our system is not without limitations. In particular, we have noted students' complaints about long autograding wait times. These complaints are due in part to the hardware configuration used by our tool; as our system was newly deployed, we hosted it on a single machine, which ran student submissions serially in FIFO order. This meant that, if two students submitted at the same time, one would have to wait for the other to finish grading. As seen in Fig. 7, this limitation was particularly evident during peak hours, when more than five submissions were made. These hours tended to be on the eve of an assignment's due date. In the case of our longest running assignment, wait times grew during peak hours as wait-times cascaded. Overall, however, we are pleased with the wait times that resulted from non-peak submissions, which were rarely greater than twice an assignment's average grading time.

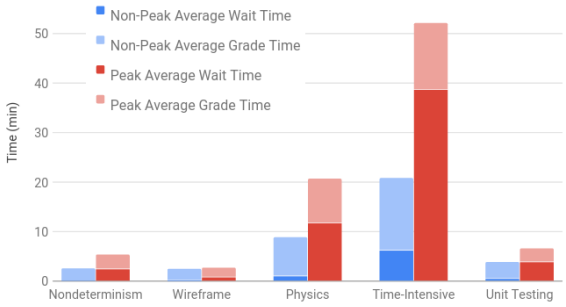

Figure 7: Wait time per assignment during peak and non-peak hours. Peak hours are defined to be hours in which 5 or more assignments were submitted. Assignment titles are given based on the titles of section 4.

6 FUTURE WORK

Autograding the computer graphics course has suggested several avenues for future improvements and research:

Sophisticated Image Comparison A simple `MSE` comparison worked reasonably well across a variety of assignments. However, it is likely that more sophisticated image comparison schemes could have better evaluated student work for partial credit.

Autograding Performance Optimization and Transparency Many students mentioned the long wait times to receive feedback on their submission. In most cases this was due to lengthy, conservative delays, e.g., to ensure completion of a ray traced rendering. As these delays are scripted, students could be forewarned about the estimated cumulative grading time. To decrease student wait times in general, we will add dependencies between testcases and adaptive termination to our system, so that submissions which fail simple testcases will not run more complicated tests.

Ease of Use Utilities At present, keyboard and mouse action specification must be done by hand. We would like to instead directly record a demonstration of the scripted sequence of actions.

System Optimization and Flexibility We plan to facilitiate element-selectors and `OCR` or image recognition solutions for GUI interaction in addition to coordinate-based mouse actions.

7 CONCLUSION

We have detailed a new system for automatically running and grading interactive computer graphics assignments within Submitty, an open-source course management and automated grading platform.

Our system allows students to verify the functionality of their work relative to the assignment specifications (and fix any errors). Furthermore, the system eliminated the need for the instructor or TA to manually test and re-run student work as part of the grading process. Student applications were given a sequence of scripted keyboard and mouse actions, and were then validated using a built-in image difference metric. Feedback from students, TA, and instructor indicate this prototype system was a success, and we plan to expand upon and improve our system going forward.

REFERENCES

[1] Christian Buhl and Fazeel Gareeboo. 2012. Automated testing: a key factor for success in video game development. Case study and lessons learned. In *Proc. PNSQC*. PNSQC. https://www.pnsqc.org/automated-testing-key-factor-success-video-game-development-case-study-lessons-learned/

[2] John DeNero, Sumukh Sridhara, Manuel Pérez-Quiñones, Aatish Nayak, and Ben Leong. 2017. Beyond Autograding: Advances in Student Feedback Platforms. In *Proc. 2017 ACM SIGCSE Tech. Symp. CSE (SIGCSE '17)*. ACM, New York, NY, USA, 651–652. https://doi.org/10.1145/3017680.3017686

[3] Michael Garland and Paul S. Heckbert. 1997. Surface Simplification Using Quadric Error Metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 209–216. https://doi.org/10.1145/258734.258849

[4] Vahid Garousi, Wasif Afzal, Adem Caglar, İhsan Berk Isik, Berker Baydan, Seçkin Çaylak, Ahmet Zeki Boyraz, Burak Yolaçan, and Kadir Herkiloğlu. 2017. Comparing Automated Visual GUI Testing Tools: An Industrial Case Study. In *Proc. 8th ACM SIGSOFT (A-TEST 2017)*. ACM, New York, NY, USA, 21–28. https://doi.org/10.1145/3121245.3121250

[5] Swapneel Mehta, Chirag Raman, Nitin Ayer, and Sameer Sahasrabudhe. 2018. Auto-Grading for 3D Modeling Assignments in MOOCs. 51–53. https://doi.org/10.1109/ICALT.2018.00012

[6] Robert Nagy, Gerardo Schneider, and Aram Timofeitchik. 2013. Automatic testing of real-time graphics systems. In *Int. Conf. TACAS*. Springer, 463–477.

[7] Alfredo Nantes, Ross Brown, and Frederic Maire. 2008. A Framework for the Semi-Automatic Testing of Video Games.. In *Proc. 4th AIIDE*. AAAI. https://aaai.org/Papers/AIIDE/2008/AIIDE08-033.pdf

[8] OpenGL. [n.d.]. *"OpenGL Homepage." opengl.org*. https://www.opengl.org/, accessed on 2018-10-13.

[9] Matthew Peveler, Evan Maicus, Buster Holzbauer, and Barbara Cutler. 2018. Analysis of Container Based vs. Jailed Sandbox Autograding Systems: (Abstract Only). In *Proc. 49th ACM Tech. Symp. CSE (SIGCSE '18)*. ACM, New York, NY, USA, 1087–1087. https://doi.org/10.1145/3159450.3162307

[10] Aras Pranckevičius. [n.d.]. *"Testing Graphics Code, 4 years later." aras-p.info*. https://aras-p.info/blog/2011/06/17/testing-graphics-code-4-years-later/, accessed on 2018-06-21.

[11] Aras Pranckevičius. [n.d.]. *"Testing Graphics Code." aras-p.info*. https://aras-p.info/blog/2007/07/31/testing-graphics-code/, accessed on 2018-06-21.

[12] Selenium HQ. [n.d.]. *"Selenium Homepage." selenium.org*. https://www.seleniumhq.org/, accessed on 2018-07-01.

[13] Selenium Master LLC. [n.d.]. *"Selenium IDE Complete List of Commands." seleniummaster.com*. http://seleniummaster.com/sitecontent/index.php/introduction-to-selenium-automation/selenium-ide/114-selenium-ide-complete-list-of-commands, accessed on 2018-06-24.

[14] Submitty. 2014-2019. http://www.submitty.org/

[15] T PLAN. [n.d.]. *"Game Test Automation." t-plan.com*. http://www.t-plan.com/game-test-automation/, accessed on 2018-06-21.

[16] Chris Wilcox. 2015. The Role of Automation in Undergraduate Computer Science Education. In *Proc. 46th ACM Tech. Symp. CSE (SIGCSE '15)*. ACM, New York, NY, USA, 90–95. https://doi.org/10.1145/2676723.2677226

[17] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In *Proc. 47th ACM Tech. Symp. CSE (SIGCSE '16)*. ACM, New York, NY, USA, 437–442. https://doi.org/10.1145/2839509.2844616

[18] Burkhard Wünsche, Zhen Chen, Alex Shaw, Thomas Suselo, Kai-Cheung Leung, Davis Muhajereen Dimalen, Wannes van der Mark, Andrew Luxton-Reilly, and Richard Lobb. 2018. Automatic assessment of OpenGL computer graphics assignments. 81–86. https://doi.org/10.1145/3197091.3197112

[19] Lisa Yan, Nick McKeown, and Chris Piech. 2017. Deep Grade: A visual approach to grading student programming assignments. In *Proc. WiCV Conf. CVPR*. WiCV. http://web.stanford.edu/~yanlisa/