Daniel Ziegler
GAM 475
Final Exam

1. **What lessons did you learn or reinforce about wrapping and abstracting systems?**
    a. **Use the file or memory system in your discussion.**

Wrapping and abstracting systems are both good ways to hide complexity from someone who will be using your wrapper. In particular, a system may be extremely comprehensive and offer *far* more tools than you or your user will ever use, let alone need.

The file and memory systems were both good examples of this phenomenon. During the course of the assignments I often found myself referring to Microsoft's documentation. Out of the countless classes and methods detailed there, I found myself using only a handful of the most basic ones. By creating a wrapper, I was effectively creating "light" versions of the memory and file system libraries—just the essentials, with no bells and whistles, and no extra methods and classes to get confused by.

2. **What did you like or dislike about test driven development (math system)?**
    a. **Comment on your testing for the PCSTree iterators**
    b. **Can this be used for every type of development (large or complex)?**

What I like about test driven development is that it simplifies the requirements of a coding project. By writing tests that specifically test the results of actions in a program, you can be much more certain that you are meeting the goals of the project. TDD also gives a greater guarantee that the project will work as intended.

The problems with test driven development aren't in passing the tests, but in making them. (It's easy to lose sight of this when the instructor makes the tests for you.) In order for test driven development to be effective, the people writing the tests need to know what to test for, and depending on the size and complexity of the code, this can be very difficult. If you're testing for a simple output given a certain input, it's easy, but the more possible use cases the program has, the larger the list of possible scenarios to test for becomes. This is why test driven development works well for smaller, more modular code, and works poorly for large frameworks. My current job doesn't use TDD, but we do write tests for all our code, and the way our work is organized lends itself well to this.

My initial experience with writing tests for the PCSTree iterator was not great. I wrote tests for all the scenarios I could think of, lost the new test files when I committed the code, and ended up failing both sets of tests when I submitted the assignment. I later went back and created far more thorough tests based on the ones in Assignment 1. Interestingly, after I finished writing and running my tests, I found there were several test where the code was working as intended, but I was testing for the wrong thing. (Specifically, the tests I wrote expected a node to remove its child nodes *and* its "next sibling" nodes upon removal, whereas in reality a deleted node only took its child nodes with it). The problem of incorrect tests is another possible point of failure for test driven development, and it's one I would not have thought of had I not written any tests myself.

3. **How do you approach a large system, its design and implementation (graphics system)?**
    a. **Given a complex system with working (albeit bad) demos**
    b. **Given that the new engineer isn't familiar with the API or still understanding and learning the material**

The graphics system gave a good example to follow for building a large and complex system using iterative development—start with something that works, and then improve on it in small increments. By working on small pieces of code, you're far less likely to break something elsewhere than if you're trying to do a lot at once. Also, when an error does happen, it's usually far less severe. Having working, possibly bad demos helps for starting the process, but if one is not available, it may also be possible to create a simple demo yourself.

Making small improvements to a working demo is also very helpful for learning a new system/API/library/etc. If "the new engineer" is unfamiliar with the material, he/she will learn much easier by tackling smaller problems, as there will be far less new material to learn and digest.

4. **What were 2 big lessons or experiences that you gained in this ridiculously arduous class (ideas: scheduling, planning, design, implementation, troubleshooting…)?**

The first lesson I learned was that time is extremely valuable. Any time you wasted is time not spent coding or planning. This especially hit home when I started a new job near the end of the quarter. Suddenly, an otherwise doable workload seemed far less so once I had eight fewer hours a day to tackle it. This also made me appreciate the importance of being engaged and attentive during this class's lectures—rewatching a lecture comes with the cost of three hours of time that you can't use. Finally, the quality of your work suffers when you're pressed for time. There were a lot of features I wanted to try and implement in my game engine that I simply didn't have time to complete. As I type this, the deadline is less than two hours away!

The second big lesson I gained was that large systems are built with small pieces. Every assignment we did this quarter was made with the intention of being used for a single system. The final project was made by making small changes, one by one. Even the spinning cube demo that we started with was very small compared to the version we had at the end of the quarter—one need only compare the number of files in each version to see the difference! I've never been very good at starting large coding projects in the past, and I think part of the reason is that I've been slightly intimidated by the scale of such projects. I think the things I've learned in this class will make me better equipped to begin and follow through with projects, both personally and professionally.