Final Exam Paper

Joey Domino

This quarter was one heck of a doozy! Never thought I'd be able to say that I made a game engine from (mostly) scratch, but here we are! I wouldn't say this quarter was as bad as optimized or making Space Invaders. In fact, I'd say Space Invaders is what really kicked my butt. However, this quarter had plenty of challenges that screwed me up for longer than it should have. Thankfully the previous courses have me very well acquainted with double linked lists, class structure, and complex inheritance to pass data to all parts of the project where required. I did have to learn and relearn how to wrap and abstract systems, fortify my skillset with test driven development, designing and implementing large systems, and, most notably, learn how to calculate frankly intense matrix calculations to create virtual shapes in a custom window and manipulate them.

The file system I created easily shows how our project handles wrapping and code abstraction. The purpose of our file system is to use Win32 file creation tools of a given type. There are two file creation options to choose from: GENERIC_READ and GENERIC_WRITE. However I also need the ability to do both, so to catch this option, I created an enumerator for READ, WRITE, and READ_WRITE selections. When we select read or the write options, the generic versions associated are selected and CreateFileA is called within the Win32 system. If the user decides to have both, we select the read_write enumerator. This selection tells the Win32 system to create a file of both GENERIC_READ and GENERIC_WRITE, allowing the created file to have both levels of access.

My wrapper file system allows the end user to more easily create files with Win32 systems with far less confusion or error. I have put in place checks to make sure whatever the user attempts to create will succeed unless bad data is put in. To allow this error system to be easily readable by the end user, I have made the wrapper method return the error code rather than the file. Instead, the file is stored in another argument passed into the method. This setup allows the end user or associated program to more easily check for error codes on creation of the file rather than needing to check if the file has proper data within it.

I'd say the biggest lesson I learned when setting up this system is to plan ahead for future errors and edge cases. It's important to plan your methods to reduce all or nearly all possible errors that come up so issues are easily identifiable. I use this style in other areas in the finished project as well. However, simply wrapping and abstracting code to handle edge cases myself isn't the only way to properly catch errors before they occur. I also implement test driven development when writing code to make sure all edge cases are handled on creation.

Test driven development is an integral part of good programming. With test driven development, the programmer is tasked with creating checks on their own code to make sure that all possible edge cases are found and caught properly. Ideally, this would mean that all programs will be submitted for wider use with no errors at all, but that's a highly ambitious goal. With test driven development, we can more easily reach that near utopia level programming if done properly.

My PCSTree double linked list system best exemplifies test driven development. The PCSTree is a binary search tree of doubly linked nodes. A node is only considered doubly linked when it is able to know who its neighbors are via pointers. In this case, each node will know who their parent is, their sibling neighbors are, and their first child if one exists. When these nodes are set up properly, we have a binary tree of doubly linked nodes that hold all required information to properly sort through the tree via pointers for a more specific guidance system.

Test driven development comes in when creating and searching through the nodes in the tree. It was my job to make sure each node is properly created, placed properly within the tree, and removed properly with all required pointers being updated with no errors. To show how test driven development is used, let's go through the creation of 3 nodes: the root node, its first child and a sibling of the child.

When first creating the PCSTree itself, the tree is empty. We then create a new node. When the new node is made, we must first check if there is a node already made. If not, then this new node is the root node. We set its pointers all to nullptr as we know the first node made will have no others to reference. Making another node will then set this one as the root's first child. We know this, because on creation we check if there's a root. In this case there is, so we call on the root node and update its child pointer to the newly created node. The new node then says that its parent is the root and the rest are null as we know no others exist. Next up is the sibling node. The root node already has its child node set, so it does not need to be updated. Instead, we tell

the child node to set its next sibling pointer to the new node. The new node then must have its previous sibling pointer set to the first child node and its parent node set to the root. And there you have it! We have successfully created 3 nodes in a PCSTree. These were all created using a large series of checks to make sure our nodes are placed in the proper locations. The checks would be part of our test driven development. We're not done yet though. Next up are the iterators.

An iterator is a different way to traverse a PCSTree. There are typically two types of iterators: forward and reverse. A forward iterator steps through each node in the tree from their creation. A reverse iterator does the exact opposite, searching through the tree starting at the last created node in the tree. This gets complicated when we have a complex tree structure where nodes have been created and removed within the middle of the tree, creating an intricate maze of pointers from node to node in almost random locations. Keep in mind that these forward and reverse iterators are a separate set of pointers, so everything stated above still applies. You can traverse the tree in 3 ways now: from the root, with the forward iterator, and the reverse iterator. These options are put in place to allow quicker traversal to find a given node in the tree.

It's pretty easy to see the positives of test driven development. Most notably, it allows for code to be far more robust from the start of the project. Edge cases are handled as production continues until no errors are possible. The largest downside to test driven development, though, is the large amount of extra code required to catch every possible edge case the system may have.

Is test driven development useful for large and/or complex systems? Sure! However, it may slow down the overall system given how big it is. The larger the system, the more checks being watched for with every new piece added to it. Test driven development is a good way to build a system if speed isn't a factor.

Building a large system requires a lot of forethought. When I build a large system, I typically look at the requirements of the system, plan how to structure them all into separate classes or scripts, and then dive deeper to decide where each piece will go within each class or script. Let's look at the graphics system as a whole to explain.

Our objective is to build a graphics system capable of creating a custom window that can display multiple complex objects built with matrix math from a custom build math library. These

shapes can then be saved to a file using the file system discussed above. We then need a system that can create textures and meshes on these matrices and display them on screen.

So, how do we begin this? Assuming we have nothing to start, my first thought would be to segment each important task to a separate class or script. So the files system, math library, GUI (also interfaces with win32 and the computer's keyboard and mouse), openGL (how we display our objects), and a wrapper to allow openGL to work with our system (SB7) all need to be placed within their own scripts and/or classes. Once created, we then interface all of these separate classes with each other to properly create the images on screen. To do this, we need a way to connect all classes and scripts together in one central location. This is where the main script, the "Game App," comes in.

The Game App is our main script where all other scripts and classes are included. Here is where we plug everything in to display and manipulate our objects on the screen, which is created within the Game App on startup using our GLFW GUI system. Now, we can create our shapes within the Game App as well, but what would be cleaner is creating a specialized method for each type of shape we want to make. This would work similarly to the file system's wrapper methods to handle Win32 file creations. Here we would set up methods to create cubes, pyramids, diamonds, and any other shape we wish to make. We could handle this in two ways: either an enumerator with a single method to create every type of shape or multiple methods where each one is specialized to create each shape.

This will segment our code and drastically increase the readability of our system. With all of these classes in place, we now have a complete graphics system that is segmented and easier to manage and read through. Having our system built like this will make it easier for future developers to dive into our code and modify it or utilize it on their own with less confusion.

Thinking back over all the work I've done for this project has me almost dizzy with all the work I put in in just 10 weeks. Honestly, we are insane. How are we still functioning after doing all this?! But here we are, still chugging along and wrapping up our final project. If I had to pick two major lessons I learned from this class and project, it's that time management and proper planning/design are integral to finishing a product relatively quickly, with robust code, and with proper documentation and implementation. I need to make sure I don't need to go back in there and rework everything I've done after all!

Time management was integral for me, because I know myself. I am one to procrastinate until I overwork myself on a task. To remedy this, I segmented pieces of a project per day. Day 1: wrap up the file system. Day 2: Tackle the math library. Day 3: continue working on the math library because it's a pain in the butt. Day 4: give up on calculating the inverse matrix in the math library and move on to the memory system. Continue this along until each piece is ready to go. Once complete, slap all those bad boys in the big project and tackle that day by day until complete. Within the final project, I would split the final to my own personal milestones. For example, I would make creating a few shapes a daily task. Then, the various camera systems can be tackled and so on until the project is complete. It's basically a faster version of sprints in an agile work environment.

Then there's the design. If I didn't first design and plan my path, I would fail this and every other class. I don't think I'd ever get anything done in my life! I need to segment my work into manageable pieces to feel like I can even manage. If there's one thing this class has shown me, it's that knowing your path forward is integral to finishing the task at hand. Otherwise, you're just running around in the dark slapping code in random places and hoping for the best.

All in all, I really enjoyed this class. I never thought I'd be able to say I made my own custom matrix math library, let alone an entire custom game engine (with a little help). While this class may have kicked my butt, I'm glad I took it. I've learned a lot that I will never forget. I look forward to using these new tools with my own projects in the future. I just hope I never need to make an entire game engine on my own. This was ridiculous building everything mostly alone! Hopefully future classes don't give me as much pause.