AOOP - Project Report

Hamed Haghjo Elias Vahlberg

May 2021

Contents

1 Introduction

2 Design

3 Testing

We tested the systems in isolation in single threaded cases not to create the illusion that all of our functions are thread safe. The methods not tested in these systems are ones that have no feasible way of causing errors in normal operating modes in single threaded operations. [Unit test of grid] [Unit test of renderer] [Unit test of GameObject and components] [Unit test of Prefabs] [Unit test of map editor] [Unit test of serialization & deserialization] *[Basic tests for multi-threaded performance]

4 Features and Problems

Our implementation of the game framework has a number of features but before mentioning those it would be appropriate to mention that the vast majority of time spent on this project was spent on tweaking and reworking core components. So that when using this framework it provides clear simple steps to construct an empty game shell while also enclosing all systems core within the actual framework. So when using this framework the only concern should be to expand it, not trying to figure out what configuration of a half baked component won't cause the framework to break.

Component system

The component system is the main way for someone using this framework to attach data and logic to an instance of a GameObject that interacts with a Grid that is then fed to the GameInstance that links this to something that is visible. [Example SimpleMove]

Serialization

Serialization is one way of saving class-based data persistently. It enables tools such as the map editor to edit a grid and then save that grid which can be loaded in by the GameInstance. The serialization target in this case is the Grid class, all objects connected to the grid that are not transient will be serialized(this is done recursively until the entire "network" of objects are serialized). Since the Grid is the serialization target it implements ways of serializing and deserializing.

This might sound like a simple feat but the 15+ hours of us working on this feature to get it to work properly says otherwise. The main reason for this is the fact that the grid contains not only GameObjects but also GameComponents within those GameObjects which can vary greatly. One default component that was particularly troublesome was the Sprite which houses a BufferedImage that is not serializable. Though the reason why it is not serializable is most likely a good one. One solution to this was to be able to re-assign BufferedImages when loading in a grid.

The method we used to solve this was to map a String reference to each BufferedImage at the time of asset loading. When loading a new Grid, remapping all buffered images requires taking the reference which is serializable and fetching the image from the GameInstance. if there are no assets loaded in the GameInstance it is a simple matter of loading the directory containing the image assets. (FOOTNOTE These assets must be the same as when the Grid was first serialized)

ObjectPrefabs are also serializable, this allows objects to be reconstructed after deserialization. One case where this would be useful is if there exists a spawner GameObject needs to be reinstated for any reason.

[Render system] From a usability standpoint the rendering of the grid is a clear time saver; it only takes a couple lines of code to instantiate and start the renderer. After that point it will run on a separate thread refreshing the view at fixed intervals updating as the grid does. This is the current code for the painting of the grid panel: [Code for grid Panel PaintComponent] Previously the stream painting of each Sprite was done in parallel, though this caused some jittering in the final view when running on Ryzen and ThreadRipper CPUs so it was omitted.

The original idea with the render system was to delegate most processing to the GPU via OpenCL. Trying to implement this rendered most of the previous code unusable. Some half-cocked C-looking code reading like an error message later (example: void JI_MatrixMul_kernel_basic(int dim, float[][] A, float[][] B)) forced the realization that it would not be feasible with our current skill set. Furthermore even the most skillfully written CUDA code would not impactfully help render a 512*512 pixel grid.

[Prefab system] ObjectPrefabs is a concept that yet again borrowed from the Unity game engine. The alternative to this would be to copy a GameObject after it was constructed or manually implementing each object statically.The ObjectPrefab system is provided as a third option for anyone hoping to make any real progress this decade. In order to create an prefab simply extend the ObjectPrefab class implement the required methods as an simple example: [Code for a Prefab] In other words ObjectPrefabs operate on the

notion that there are only so many meaningfully different ways that you can implement a 2D box. Not all objects are all unique, in fact most objects in games are not unique at all. Take for instance the this implementation of wall in Sokoban: [Wall prefab] Since this framework does not have to be written in x86 assembly and fit on a box of punch cards totaling 2KB; each such non-unique object is constructed in much the same way but are stored as separate objects. Though the main purpose of the implementation of this system is to organize GameObject construction instructions speed up development time and.

[Map editor] The map-editor was an early product that we chose to implement to maintain a high efficiency when testing the framework. The map-editor was created in such a way that it runs on a separate thread from the instance of the game, but it shares the same grid. This would help the game-developer using the framework to be able to run several different tests and not worry about adding separate code sections for setting up each test. The only setup would be to add the needed prefabs to test into the map-editor. The map editor is also made for the game-developer to serialize the grid when for instance he's creating new "levels" for the game.

[UI and event management] [Threading] Threads are good because they're like strings within java. So creating new threads would be only to create new strings within the code. [Event system] *[Sound system]

Like all projects this one accrued a significant amount of technical debt. This was mainly due to falling into many of the common AntiPatterns being to admit these might help understanding some of the less understandable design choices. The most common and prevalent of these was Band Aid/The Quick Fix. "There is nothing more permanent than a temporary solution" Another prevalent opposing force was Feature Creep playing on the programmer ego. Construction workers know better than to install expensive interior decoration before the roof has been put up. "Sometimes there is little distinction between what's needed and what's 'neat'." As mentioned earlier

As mentioned in the design section the God Class problem was also somewhat present and recurring when there were disputes in placement of code.

When trying to design code it is also easy to "Design For The Sake Of Design" forgetting that it not only needs to look pretty it also has to work (in some capacity at least).

One problem that arose often regarding the MVC was the "Doer And Knower" antipattern. Where one class has the operations needed to be done but no state connected to the result or any external factors that could cause the operations to fail. This creates coupling between classes where it is not needed.

5 Results

6 Version Control System

We have worked with version control systems previously, *Git* mainly, so we have experience in this field. When working on the project we used the git repository hosting service *GitHub* to maintain our common repository. We used the Microsoft Windows environment application *GitBash* because we are used to its error logs and we felt like we could dive more into detail with what possibly could go wrong. Mainly because we just switched to the IDE *IntelliJ* which has a different layout and handling algorithm when communicating with our repository on GitHub than previously used IDE's such as *VSCode*.

We tried to use a lot of the functionalities of GitHub to make sure our project link at GitHub maintained a professional look. We used the functionalities of Milestone/Issues/Tags/Releases/Git-pull requests and so on. Even if we did not really need most of it due to the fact that we mainly worked on the project using the new functionality of our common IDE - IntelliJ called *CodeWithMe* (CWM). Where we could work together with the project in real-time, leaving us with less tendency of the very popular GIT merging conflicts. We were sometimes working on the same issues, and we felt like CWM helped us avoid these potential conflicts. If we were not using CWM, we'd assign ourselves with different issues and branch out of the main repository to later merge together.

7 Sources