**Spike:** Task 31

**Title:** Custom Project

**Author:** Sam Huffer, 101633177

# Custom Project Plan

This year as part of the capstone units for my BA. Games and Interactivity, along with the rest of my teammates in Under Ctrl, I have contributed to the development of the game *Get the Fog Out*, a small-scale RTS where you seek to repair your ship and escape a desolate planet before a hostile fog can kill you. I was one of the programmers for it, programming the fog, tutorial, and dialogue system.

For my custom project, I plan to dissect scripts related to those areas in a post-mortem of my contributions to *Get the Fog Out*. Referring to data structures and software patterns mentioned in the lectures, I shall go through those scripts and identify where particular structures and patterns have been used currently or in the past, how they were implemented and why, and - knowing what I do now thanks to this unit - whether those choices of structures, patterns and their implementation were optimal, why or why not, and any improvements I would make were we to recreate *Get the Fog Out* again from scratch or further develop it after submission. I do not plan to cover our design decisions or the rationale behind them and any changes I’d make there except as they relate to choices of data structures, patterns, or pattern implementation.

# Intended Learning Outcomes

1. **Design**: Discuss game engine components including architectures of components, selection of components for a particular game specification, the role and purpose of specific game engine components, and the relationship of components with underlying technologies.
2. **Implementation**: Create games that utilise and demonstrate game engine component functionality, including the implementation of components that encapsulate specific low-level APIs.
3. **Performance**: Explain and illustrate the role of data structures and patterns in game programming, and rationalise the selection of these for the development of a specified game scenario.
4. **Maintenance**: Explain and illustrate the role of data structures and patterns in game programming, and rationalise the selection of these for the development of a specified game scenario.

# Links

* *Get the Fog Out* Build: <https://drive.google.com/open?id=1g-kbjxa0kVx5RDpjsmL9vFlKOIz0-2NH>
* *Get the Fog Out* Source Code: <https://drive.google.com/open?id=1RWaMqF8KwiObDIKwVbxbsrx3JdWmEQL7>

# Fog

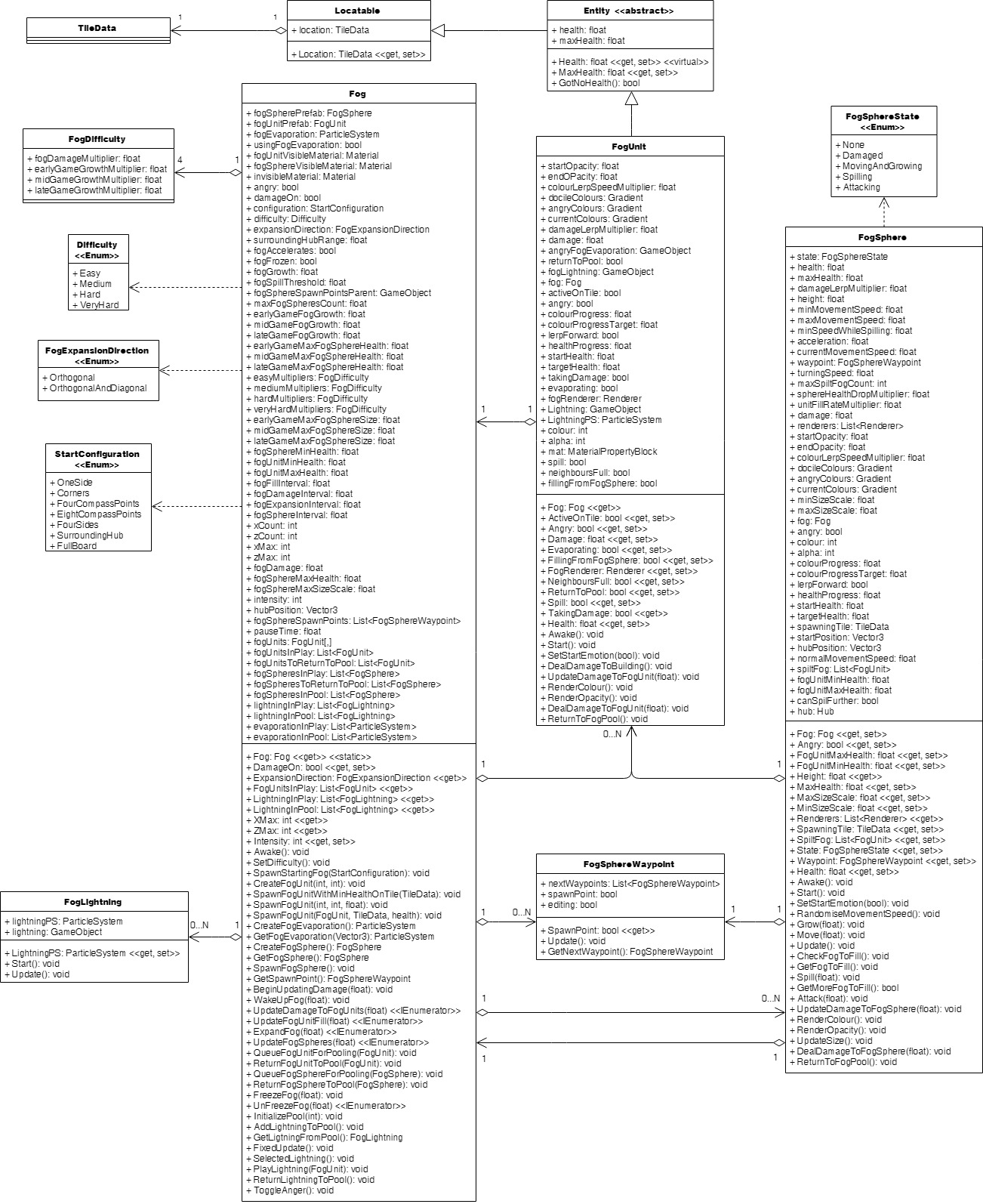


Figure 1: the current structure of the fog-related classes.

## Data Structures

### FogUnits

Figure 2: Fog.UpdateFogUnitFill() under the profiler while all FogUnits were being updated every time the method was called.

Originally, about 2500 FogUnits in Fog were split between two lists, one of the FogUnits currently in play, and the other of FogUnits that had been pooled, with FogUnits being allocated to TileDatas at runtime and drawn

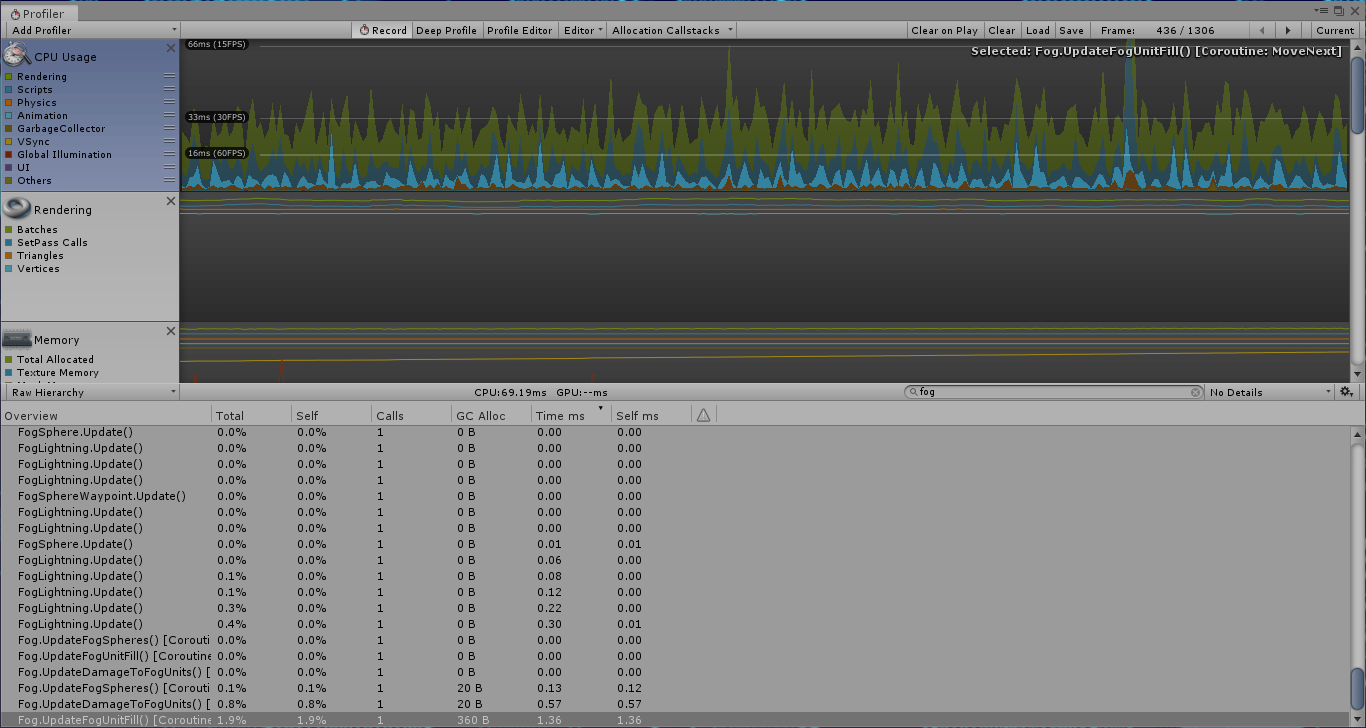
from the FogUnit pool, and put back there when destroyed. The result: a very slow game. Even running update methods less often than every frame (e.g. every 0.25 seconds for UpdateFogFill()) still had regular and noticeable drops in performance (fig. 2), which became more prominent after we increased the map size from 51x51 (2601 FogUnits) to 71x71 (5041 FogUnits). Eventually, we tried a combination of a) storing all FogUnits in a 2D array, with FogUnits’ positions in the array matching their corresponding TileData’s position in the 2D array of TileDatas, and accessing the array whenever interacting with a specific FogUnit of a known position (e.g. to check if it was active), b) removing the fogUnitsInPool List but still keeping all active FogUnits in the List fogUnitsInPlay for processing all active FogUnits one after the other, and c) spreading out the updating of FogUnits as part of UpdateFogUnitFill() over all frames in the 0.25 second interval, rather than updating all of them within the one frame. The combined results of these decisions led to a much smoother framerate, with minimal spikes in resources usage and no noticeable drops in frame rate attributable to FogUnits (fig. 3). If I were to do this project again, I would want to use this setup right from the start, especially if I knew there would be thousands of FogUnits. If the number was much lower, a List might be okay performance-wise, but this would still be preferable.

Figure 3: UpdateFogUnitFill() as an IEnumerator updating chunks of FogUnits each frame over the whole 0.25 second interval, rather than updating all of them within the one frame.

### Fog Spheres

From their addition until the final submission of *Get the Fog Out*, FogSpheres were stored between an in-play List and a pooled List, with FogSpheres swapped between them at runtime as they were brought into play or died. When I was adding the 2D array for the FogUnits, I didn’t feel it would be helpful or necessary for the FogSpheres to be put in a 2D array since a) FogSpheres weren’t tied to particular tiles or positions and b) there were only a dozen or so of them, so searching through lists to check if they were active or not wouldn’t be particularly time consuming. (If there were going to be hundreds or more, I’d only add a bool flag to FogSpheres, as concern a) would still be a consideration.) Managing the FogSpheres between these two lists proved simple and elegant, with negligible effects on performance thanks to their low number.

### Difficulty modifiers

I stored the modifiers for various floats at a given level of difficulty in a struct with appropriate fields. I found this solution to be clean and the structs (once created) to be interchangeable, as the appropriate struct merely needed to be accessed to determine values for that difficulty when the difficulty would be set.

### Fog sphere spawn points

The FogSphere spawn points I stored in a List of FogSphereWaypoints (as the spawn points are just FogSphereWaypoints with the “spawnPoint” bool checked in the inspector), allowing for flexible selection of any FogSphereWaypoint from the List, seeing as it’s only used when picking a random FogSphereWaypoint for a FogSphere to spawn at.

## Design Patterns

* State pattern: not used explicitly, but fog and fog units and fog spheres have stages of gameplay that progress as the game progresses; perhaps using structs with all those shared values that can be plugged in and swapped out when the fog’s state changes
* Strategy pattern: not used, but might be usable with fog unit/sphere behaviour, particularly for fog spheres which go through different concrete states encapsulated in a handful of methods that could be isolated in a strategy class and plugged in as necessary.
* Singleton pattern: used to allow referencing of the fog script from other scripts without needing it as a field of that script, and to ensure there’s only one fog; if several were running, performance would tank
* Command / Message pattern: not used, but could have benefit in future iterations for prompting the fog to do particular things, such as spawn, wake up, increment difficulty, etc. by just issuing a command to the one method rather than going through custom methods for each
* Factory pattern: informally implemented with putting inactive fog units back into the scene, configuring them properly. Also done with the same with fog spheres, and a similar thing with the setting up of the fog generally. If redone, might want to split the factory methods out into a separate fog factory class to clearly separate different areas of functionality.
* Component pattern: used by default as part of unity with game objects and their components, like the transform and renderer components; the fog unit and fog sphere scripts also serve as components for their respective game objects.

## Usability Patterns

* State: partial informal implementation; when the fog goes from asleep to awake, it changes colour, and again when it goes from its second intensity level to its third intensity level. However, no similar obvious change is made for the zero-th to first and first to second intensity changes.
* Progress: informal implementation with the fog encroaching on the player vs the player pushing it back; its closeness to the player and how far out the player has expanded offer some indication of progress.

## Future Changes

* Split fog into fog factory for fog unit and fog sphere instantiation, fog for updating the fog units and spheres, and a fog lightning storm class for updating the fog lightning (would require Fog to have a GetFogAtPos(int, int) method or something as the fog lightning content checks if fog units are active on tiles before playing.
* Remove FogSphere members UpdateDamageToFogSphere() and DealDamageToFogSphere(), as the latter is never used.
* Implement component pattern further with having common components between fog spheres, fog units, and other classes in their inheritance tree as appropriate to cut down on class size

# Dialogue System

## Data Structures

## Design Patterns

# Tutorial

## Data Structures

## Design Patterns

Screenshots and diagrams to back up

Make speculations on future design that relate to the four learning outcomes, back up with discussion of patterns to use and graphs and charts of the design

Email Tien with more complete draft on doubtfire, sort out feedback then

Fair game to list by tasks in LSR and address outcomes they address, rather than by outcome and listing the tasks they address