**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.

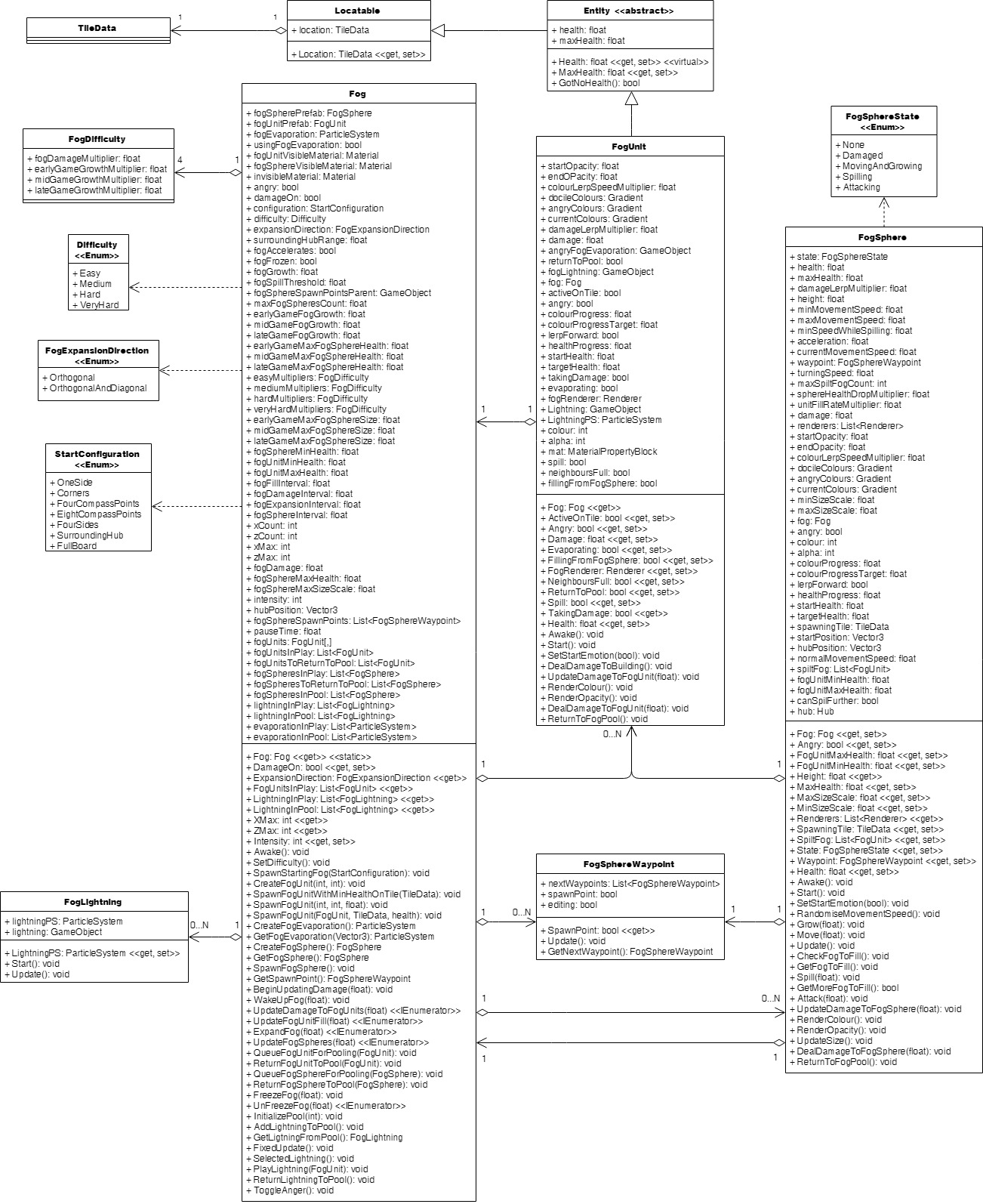
For my custom project, I plan to dissect scripts related to those areas in a post-mortem of my contributions to *Get the Fog Out*. Referring to data structures and software patterns mentioned in the lectures, I shall go through those scripts and identify where particular structures and patterns have 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.

# Fog

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



## Data Structures

### FogUnits

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.

* Fog units: ~2600
  + Were in list with fog units allocated to tiles at runtime, pooling in a separate list when destroyed and pulled from the list when re-instantiated.
    - Results: very slow
  + Spread out updating of units over intervals for different update areas, i.e. rather than calling their updates every frame, they’re all called once every 0.25 seconds or something.
    - Results: faster, but still with noticeable drops in performance every few seconds
  + 2D array with all fog units, fog units matched 1:1 to tiles with positions in array matching positions in tile. Lists retained to list all fog units currently in play. 2D array accessed when interactions with specific fog units of known positions are required (e.g. checking if they’re active), List when processing all fog units in play. Also spread out updating over a number of frames rather than trying to do it all in one frame,
    - Results: much faster and smoother, with minimal spikes in resources usage; no noticeable drops in frame rate attributable to fog units. If done in future, might use this setup from start if I know there’s gonna be a ridiculous number of fog units; if the number is relatively low, this would still be quicker but a list alone would still be fine performance-wise
* Fog sphere data structure; ~dozen fog spheres total
  + Pool and in-play lists, spheres moved between lists at run time as they are instantiated or die
    - Results: simple and elegant, negligible effect on performance
* Difficulty modifiers
  + Modifiers per difficulty stored in a struct with float fields; structs for each of four difficulties stored in own fields
    - Results: clean and interchangeable, when difficulty is set, can just get all the values from a single struct and be done with it.
* Fog sphere spawn points
  + List of FogSphereWaypoints
    - Results: flexible selection of fog sphere spawn points from the list; don’t care about them being locked to a particular position, just care about them being valid; if they need their tile, the waypoint has the reference to the tile.

## Design Patterns

* Composite pattern: fog as a whole comprised of fog units and fog spheres
* 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