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 team-mates 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 planned 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 planned to go through those scripts and identify where particular structures and patterns had 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.

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.

Notes

- Within the repo, I have added a zipped build of *Get the Fog Out* and a zip file with all of the scripts used. Of the classes that comprise *Get the Fog Out*, my contributions were:
    - Most of Fog.
    - Most of FogUnit.
    - Most of FogSphere.
    - All of FogSphereWaypoint.
    - Some of Entity.
    - All of Locatable.
    - All of FogDifficulty.
    - All of DialogueBoxController.
    - Most of TutorialController.
    - Some of ObjectiveController.
    - All of btnTutorial.
    - All of CameraInput.
    - Most of DialogueBox.
    - All of DialogueSet.
    - All of ExpressionDialoguePair.
    - All of ColourTag.
    - Most or all of each of the enums listed in the UML diagrams below.
    - Some of / a little of / none of each of the other scripts or classes.
- *Get the Fog Out*'s itch.io page, which has further information, team credits, and a build available for download, can be found here: https://underctrlgames.itch.io/get-the-fog-out
- I have uploaded *Get the Fog Out*'s Unity project, which contains all of *Get the Fog Out*'s assets and source code, here: https://drive.google.com/open?id=1RWaMqF8KwiObDIKwVbxbsrx3JdWmEQL7

Fog

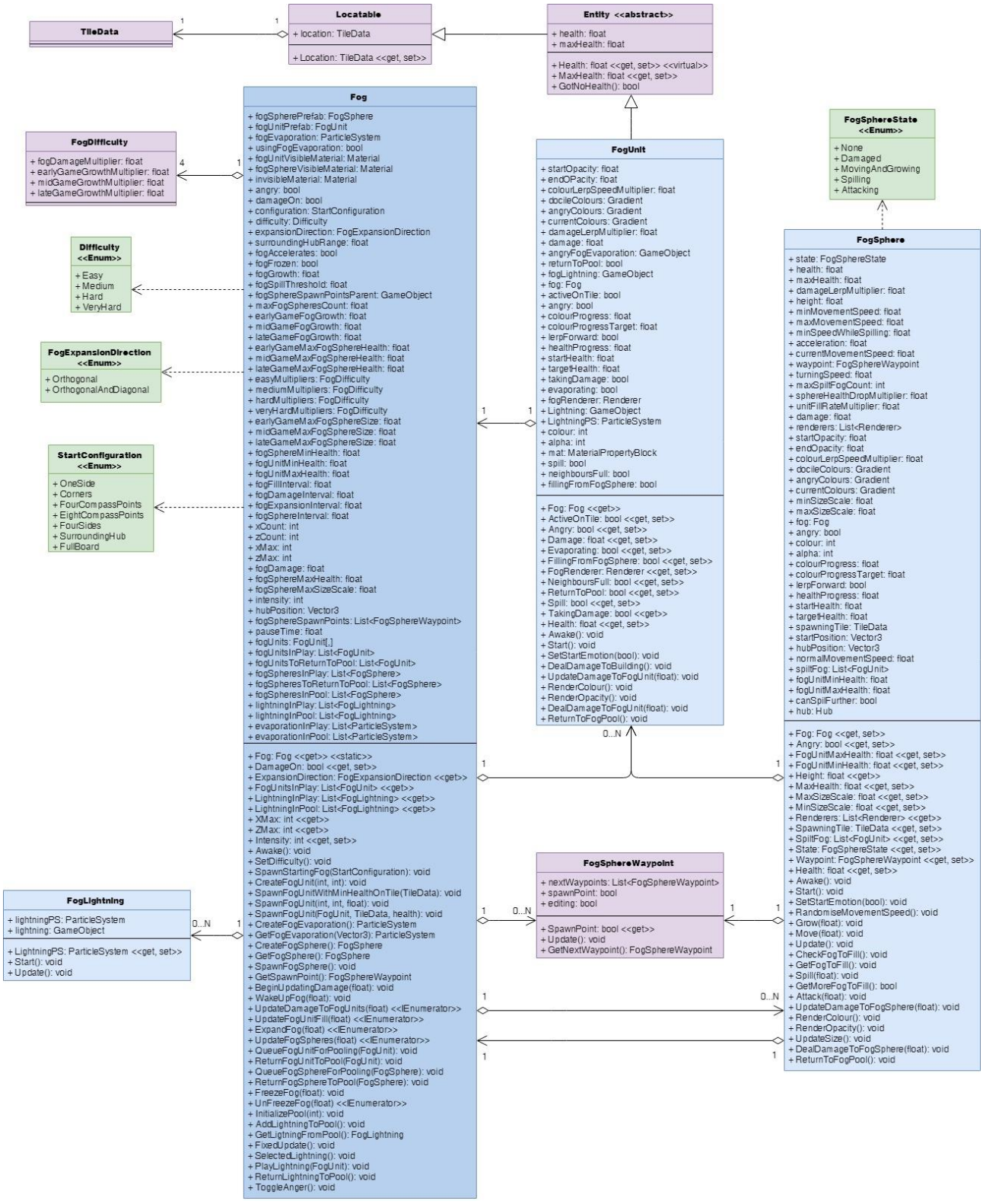

Figure 1: the current structure of the fog-related classes.

Data Structures

FogUnits: List, 2D Array

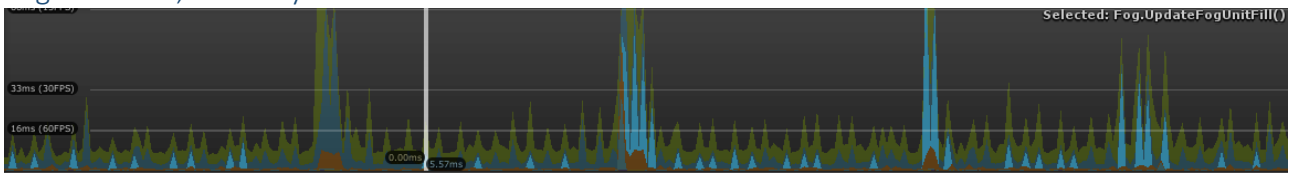

Figure 2: Fog.UpdateFogUnitFill() under the profiler while all FogUnits were being updated every time the method was called.

Originally, 2601 FogUnits were split between two lists in Fog, one of the FogUnits currently in play, and the other of FogUnits that had been pooled, with FogUnits drawn from the FogUnit pool and allocated to TileDatas at runtime, 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 left 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

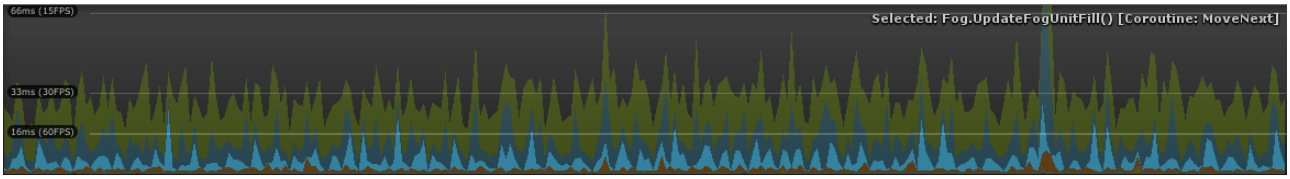

Figure 3: UpdateFogUnitFill() as an IEnumerator updating chunks of FogUnits each frame over the whole 0.25 second interval, rather than updating all of them within the one frame.

(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.

Fog Spheres: List

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: Struct

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.

FogSphere Spawn Points: List

The FogSphere spawn points I stored in a List of FogSphereWaypoints (as the spawn points were just FogSphereWaypoints with the "spawnPoint" bool checked in the inspector), allowing for flexible selection of any

FogSphereWaypoint from the List, seeing as they were only used when picking a random FogSphereWaypoint for a FogSphere to spawn at.

Design Patterns

State Pattern

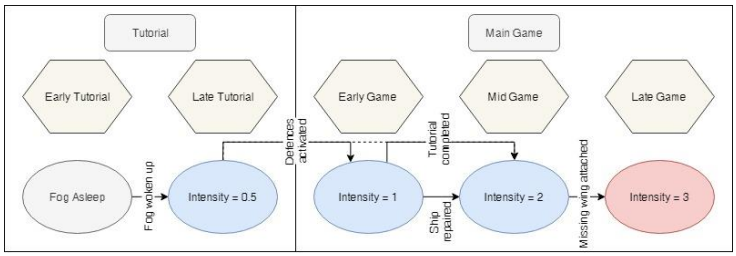

Figure 4: the informal "stages" of the fog.

```
switch (intensity)
{
    case 0:
        fogGrowth = easyMultipliers.earlyGameGrowthMultiplier * 0.5f;
        fogSphereMaxHealth = earlyGameMaxFogSphereHealth;
        fogSphereMaxSizeScale = earlyGameMaxFogSphereSize;

        if (angry)
        {
            ToggleAnger();
        }

        break;
    case 1:
        fogGrowth = earlyGameFogGrowth;
        fogSphereMaxHealth = earlyGameMaxFogSphereHealth;
        fogSphereMaxSizeScale = earlyGameMaxFogSphereSize;

        if (angry)
        {
            ToggleAnger();
        }

        break;
    case 2:
```

Figure 5: an excerpt from Fog.Intensity.

The state pattern was informally used with the Fog and FogUnits, but more explicitly implemented in FogSpheres. In the case of the informal implementation, Fog has an integer Intensity property (fig. 4) that updates the rate of growth for FogUnits and the maximum size and health of FogSpheres (fig. 5). As all the values updated are stored in Fog itself, this seems a clean enough implementation to retain in future barring other extensive restructures.

More formally, the FogSpheres have distinct states of behaviour corresponding to values of the FogSphereState enum; during UpdateFogSpheres(), Fog runs run the FogSphere's state through a switch and calls FogSphere's methods as appropriate for that state. Each executes its behaviour and checks if conditions for updating the FogSphere's state are satisfied (fig. 6). It works cleanly enough, but leaves room for improvement that I will discuss below.

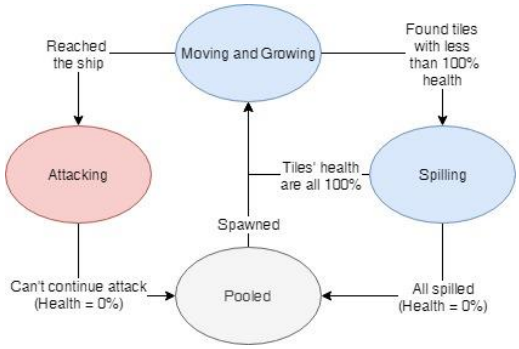

Figure 6: FogSphere's states.

Singleton Pattern

The singleton pattern was implemented extensively throughout *Get the Fog Out* to allow access to various controller or manager classes from anywhere that needed it without having to have it be a field of the class. Fog is one of those manager / controller classes, controlling the FogUnits and FogSpheres, so its Instance property and Awake() method were programmed to implement the singleton pattern as would best fit implementation in Unity. The other effect of ensuring only one instance of Fog can exist at a time was also important; given the high number of FogUnits, a duplicate Fog with its own FogUnits would cause performance to drop unacceptably. As such, this is an aspect of Fog that I would not alter in any future updates to *Get the Fog Out*.

Factory Pattern

Though not split into a separate class, the Fog class features factory pattern methods for standardised creation of FogUnits and FogSpheres when the fog is initialised, and for instantiation / activation of FogUnits and FogSpheres as they are spawned. Programmatically, this helped keep Fog's code clean when it came to creating or spawning bits of fog, as only one of a handful of methods would need to be called. However, the inclusion of those methods in Fog rather than a dedicated FogFactory class wasn't an ideal choice, as it resulted in the class being much longer (and that much harder to search through) than it could have been.

Component Pattern

While I did not implement the component pattern with the Fog, FogUnits and FogSpheres, Unity's GameObjects do make use of it with components like Transforms and Renderers, thereby making it a part of how they work by default.

Usability Patterns

State / Progress Patterns

Alongside the informal programmatic states of the fog discussed above, the state and / or progress usability pattern was also informally implemented through the changes in colour applied to FogSpheres and FogUnits at different points (fig. 7): when the Fog wakes up, it changes colour from grey to blue and grey, and again from blue and grey to red and grey when the player attaches their ship's missing wing to the rest of their ship. These changes in colour correspond to and convey to the player changes in the Intensity of the Fog's behaviour due to the player's progress through the game: from asleep to awake and approaching, and from approaching slowly to more aggressively.

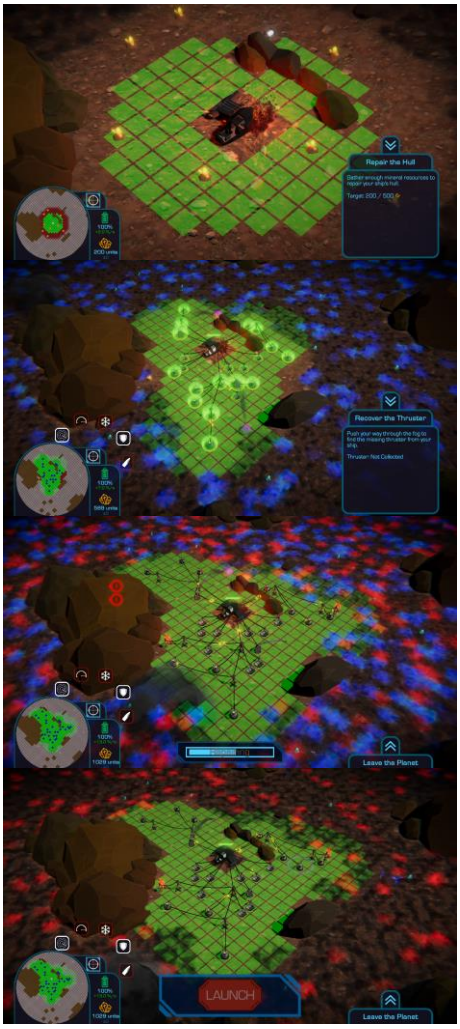

Figure 7: the progression of the colour of the fog, from a black-ish grey (asleep) to blue (awake) to red (the final stage of the game).

Restructure

Were I to restructure, update or redo the fog classes, there are a number of changes that I would make. There are a number of methods and options that were made available but never made use of (e.g. FogSphere's UpdateDamageToFogSphere() and DealDamageToFogSphere() methods, and the orthogonal-only FogExpansionDirection value and the corresponding code in Fog.ExpandFog(). Where they would not be required or were obsolesced by design decisions made for *Get the Fog Out* and would not be made use of again, I would remove them. Though the savings in compilation time, speed at runtime, and file size would be negligible, they would be savings nonetheless. More substantially however, removing that unnecessary content would reduce the methods, lines of code, etc. that developers would have to search through to while fixing bugs and updating code.

On that same note, the Fog, FogUnit and FogSphere classes are rather long, especially the former. Fog and FogSphere have members that could be removed and turned into their own classes, while FogUnit and FogSphere feature methods that are near identical, yet they don't share a common ancestor in the inheritance hierarchy or make use of common components, resulting in duplicated code. To address these issues in a further iteration of *Get the Fog Out*:

- I would break Fog up into manager classes for FogUnits, FogSpheres and FogLightning, as well as a factory class that could instantiate and do basic configuration of each of those parts of the fog for their respective manager classes. Each manager class and FogFactory would be a singleton to ensure their uniqueness and universal access to them.
- I would dismantle FogUnit's inheritance structure (FogUnit → Entity →Locatable), and instead use those separate classes as components of FogUnits. In the case of Entity, this would boil it down to just a Health component that FogSphere could also use and remove some of FogSphere's methods that a Health component could take care of. FogUnit and FogSphere's common rendering members could also be extracted and made into a separate class for use as a component of FogUnit and FogSphere.

- I would amend *Get the Fog Out*'s current implementation of the state pattern and go for a more modular state pattern approach, splitting the methods associated with each of FogSphere's states into separate classes that operate on FogSpheres and inherit from a common base state class, with instantiated objects of those child classes able to be plugged in interchangeably as FogSpheres' current states.

While breaking up these classes like this would create a more intricate network of which class uses what and who contains copies of who, and the more prevalent use of public properties this would necessitate might slow down the game (albeit negligibly), it would also result in smaller versions of the existing classes that have more clearly defined and singular purposes, those classes being more readable and more easily searchable during debugging, and bugs more pinpoint-able to particular classes rather than somewhere in a mega-class, therefore making them more manageable and easier to maintain.

For further iterations beyond this restructure, I would want to look into further optimising the FogEntityRenderer and Helath components using the flyweight pattern of splitting classes into shared and unique information, as a lot of the information in that class will always be the same across all FogUnits and across all FogSpheres, and updating the single copy everyone uses would save a lot of time.

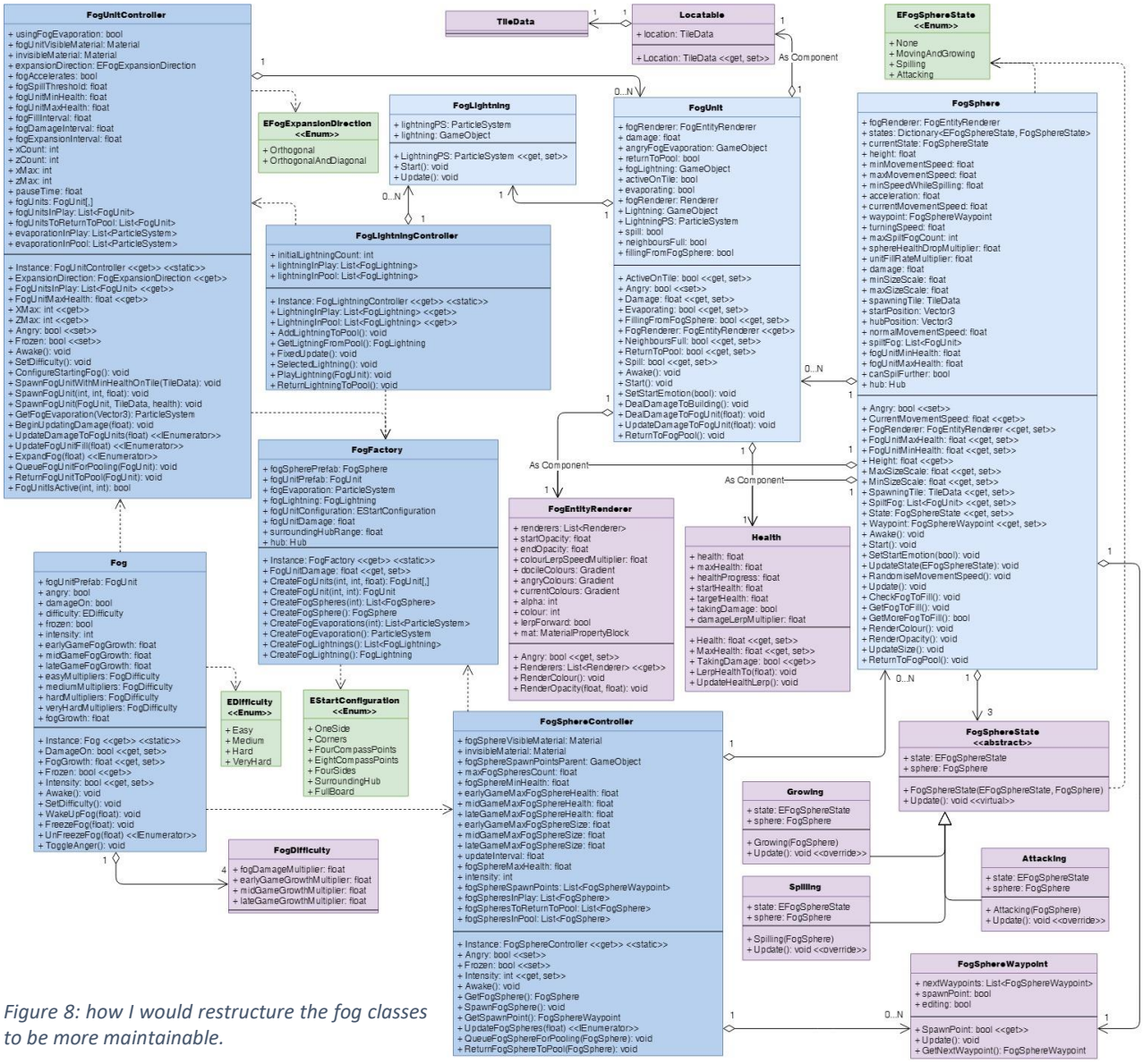

Figure 8: how I would restructure the fog classes to be more maintainable.

Objectives (Tutorial / Main Game)

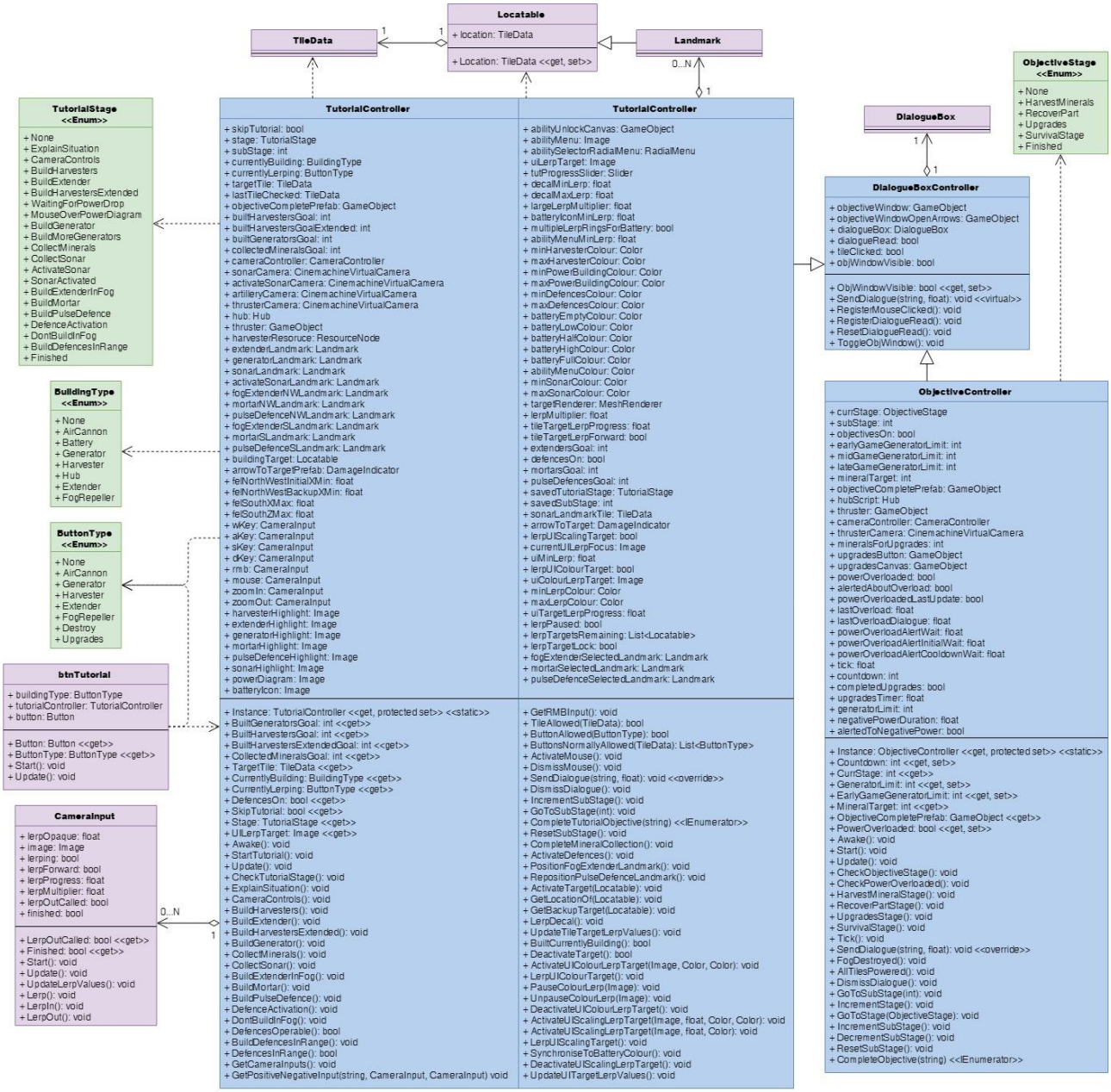

Figure 9: the current structure of the tutorial-related classes.

Data Structures

Lerp Targets Remaining: List

Surprisingly, TutorialController only has one field that is a collection of items: lerpTargetsRemaining, the list of the Locatables to which upcoming tutorial stages will move the lerp target in order to prompt players to do something at that location (fig. 10). I stored them in a list to allow them to be added and removed quickly and painlessly. Searches of the list to check if a Locatable is still awaiting use only have to run through a relatively small number of elements, so any loss of speed from using this over another data struc-

Figure 10: the activate sonar stage of the tutorial; the player is zoomed out and being prompted to activate the sonar (whose area of effect is yellow) within the lerp target (blue).

ture that would be more efficient is negligible, particularly compared to the ease of insertion, removal and searching. If a more efficient but equally or more easily usable data structure were available, it might be worth switching to that, but for the moment Lists are perfectly adequate, so I would leave this choice of data structure as is.

ButtonsNormallyAllowed: List

The method ButtonsNormallyAllowed() compiles a List of buttons that would normally be operable when interacting with a particular tile. It compiles it as a List at runtime, and which buttons are usable on that tile aren't guaranteed, so having an array of a fixed length would be less appropriate. Using Lists also affords the flexibility and ease of addition and removal I mentioned above. The choice of data structure, I wouldn't change.

Structurally, however, it's arguably more appropriate for this to be a member of the TileData class rather than TutorialController. If it were, anything that needed to check what would normally be allowed would only have to access the TileData, not pass it to TutorialController and wait for it to do the processing. (Anything assessing what buttons would *currently be* allowed would still have to go through TutorialController.) Were it done this way, I'd have it check if it had previously compiled the List, compiling it and storing it if it hadn't before returning the stored List. (To avoid tiles with no ResourceNodes only allowing Harvesters to be built after their ResourceNodes were fully mined and their corresponding Harvesters dismantled, adding a bool flag to indicate the button list is out of date would be necessary.) Compiling the list on request in the TileData itself wouldn't increase the burden on the computer, as it's already compiling these lists at runtime, and would keep the initial load time down as 5000+ TileDatas wouldn't be all processing information at the start. Storing the list upon compilation and merely retrieving it upon subsequent requests would improve efficiency as the list wouldn't be recompiled every time it gets requested, only being compiled once, twice at the most if that tile had a ResourceNode.

Local Data Structures of Methods: List

TutorialController's methods DefencesOperable(), PositionFogExtenderLandmark() and GetBackupTarget() also make use of local Lists in their functioning, each compiling them to search (DefencesOperable()) or compare with another List (PositionFogExtenderLandmark() and GetBackupTarget()). In each case, Lists were chosen because they facilitate simple and flexible addition to, removal from, and searching amongst their elements while the containing method determines its outcome. Again, I would leave the choice of data structure alone unless a faster but equally easy to use data structure was made available.

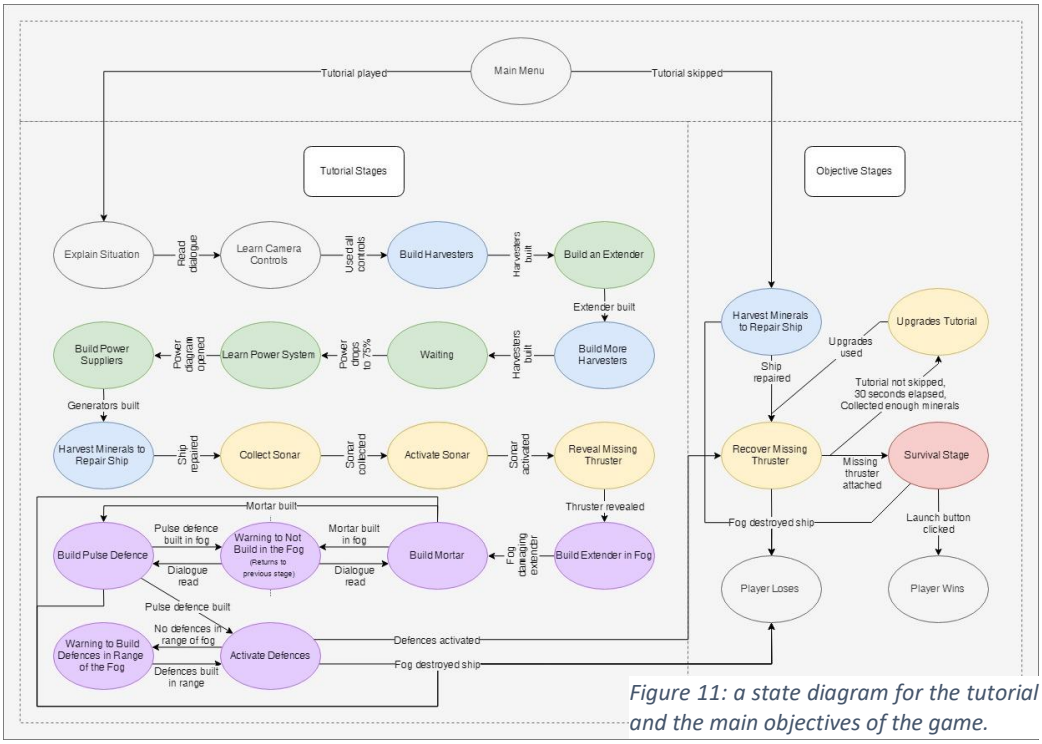

Figure 11: a state diagram for the tutorial and the main objectives of the game.

Design Patterns

State Pattern

Get the Fog Out makes very extensive use of the state pattern with regards to stages

of the game, as players have to complete a series of objectives throughout the tutorial and during regular playing of the game (fig. 11). It was implemented in the TutorialController and ObjectiveController classes using a switch in their Update() methods to check an enum field for the current stage, then call the corresponding method. That method in turn encapsulates another switch for the current point within that stage, with cases outlining what should happen at that substage and what conditions there are for proceeding to the next stage or substage. When the tutorial is completed or skipped, TutorialController tells ObjectiveController to start going through its stages and progresses itself to the Finished stage (i.e. does nothing more). This switch-based implementation was chosen because it was simple, straightforward and got the job done. However, each stage's related methods are encapsulated within TutorialController and Objective controller, leaving them with more lines of code than might be most easily manageable. The splitting of stages between TutorialController and ObjectiveController was originally done by accident as we were dividing up work to be done and different people worked on different parts of the game, but from a programmatic and structural standpoint, it lead to some unnecessary duplication and complexity that could have been avoided had the two classes just been the same. Future iterations could restructure them substantially, which I shall address below.

Singleton Pattern

TutorialController and ObjectiveController both had the singleton pattern implemented, the same as the Fog, to ensure that there would only be one copy of each and that everything could access it as they required. Each had it implemented it separately rather than as part of their shared DialogueBoxController parent class to avoid complications from their being two DialogueBoxControllers, from their instances being DialogueBoxControllers rather than Tutorial or ObjectiveControllers, or from having to cast them as their child classes every time they were accessed, whether directly or via an Instance property that hides another inherited Instance property.

Observer Pattern

How the btnTutorial and TutorialController classes interact could loosely be described as an inverse implementation of the observer pattern. The normal pattern of the observer pattern is that observers subscribe to be alerted when a subject does something. Here, when the game starts, the btnTutorials get a reference to TutorialController, then access its public members to check every time their Update() method is run if the associated Button object should be interactable or not. This implementation was used because it was simple and effective enough that we could set it up and work on larger parts of the game. However, it is less efficient than a proper observer pattern-based implementation, as it checks every update if its state should change rather than being occasionally notified when it should change its state. That extra workload for the computer is minimal as there are only a handful of btnTutorials in the scene, but it is still workload every frame that nonetheless could be removed.

Usability Patterns

Focus Pattern

The tutorial makes extensive use of the focus pattern in teaching players how to play *Get the fog Out*. The tutorial uses diagrams and lerp targets in the UI and on the map to draw the player's attention towards whatever the tutorial is trying to teach the player about and which they therefore need to interact with (fig. 10, fig. 12). The tutorial's use of it also restricts player input to the UI element or feature that players' attention is being drawn to, forcing

Figure 12: examples of the focus pattern throughout the tutorial. L-R: prompts for the player to a) move the camera with the WASD keys and b) dragging with the right mouse button, c) within the build menu, the building that is the focus of the current stage being highlighted and fully opaque while others are dimmed and transparent, d) a lerp target in the UI to draw the player's attention to the ability menu, and e) a marker pointing towards the lerp target since it is in the player's peripheral vision.

them to use it to progress. This ensures don't just read or hear about feature X but use feature X to build their first-hand understanding of it.

Magnetism Pattern

The tutorial features several manifestations of the magnetism pattern. Whenever player is prompted to build a building, they are able to build it on any tile that it would be valid to build that building on. Once the player reads through all the dialogue prompting them to do that, however, a lerp target appears around a chosen tile, not only focusing players' attention on that tile but restricting players to building only on that tile. This is done to help reinforce which tiles the current type of building can be built upon, and prevent beginners selecting tiles that they cannot build the current building on.

Furthermore, whenever an already occupied tile would have a lerp target displayed on it, the tutorial goes through the nearby tiles, finds one that's unoccupied and not intended to be used later in the tutorial, and selects that tile to be used instead of the original but occupied tile. This prevents the undesirable scenario of the tutorial prompting the player to build where they cannot, guiding them towards unoccupied, acceptable tiles.

Another instance of the magnetism pattern that is similar in subtlety and function occurs in the stage where the player places an extender in the fog. To the west of the player's ship are a mound of rocks (fig. 10) and then the missing thruster. Extending out to it requires going north or south around the rocks. When the player gets to this stage, they will have built other extenders that can further be extended from to go around the rocks, but which could lead north *or* south. So, the tutorial checks all powered tiles within accepted ranges, sees whether the powered tiles extend further north or south around the rocks, and prompts them to build the extender in the fog in that direction. If they're equal, it picks at random. If no tiles in the accepted ranges are powered and covered in fog, the tutorial prompts the player to first build an extender outside the fog in order to power tiles within the fog for building upon. The direction selected here also affects where the player will later be prompted to build their defensive buildings such that they expand outwards and start clearing fog in the same direction.

State Pattern

TutorialController and ObjectiveController feature several manifestations of the state usability pattern to indicate changes in options or status. As discussed above, the fog changes colour as it becomes more aggressive, going from grey and asleep, to blue and active, to red and aggressive (fig. 7), visually representing how players might expect the fog to behave.

btnTutorial makes buttons interactable or not according to what whether that button would normally be available on that tile and according to whether TutorialController says it should be available or unavailable at any given point in the tutorial. Available buttons have full opacity and may display information and highlight themselves when moused over; unavailable buttons have half opacity and are greyed out (fig. 12c). This communicates to players what they can and cannot do at a given point in the game, as well as which options the game wants them to use in the case of occurrences during the tutorial.

When the player's objective changes upon the beginning of a new stage of *Get the Fog Out* or its tutorial, the objective window (fig. 13) tweens onto the screen and updates its text for the current objective. The methods for the tweening and the updating of text are called at TutorialController and ObjectiveController's discretion, but are members of UIController. TutorialController and ObjectiveController also call UIController's methods for displaying large buttons to complete major stages in the game (fig. 14).

Figure 13: the objective window.

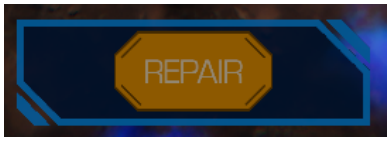

Figure 14: the repair button that displays when players have sufficient minerals to repair their ship.

Automatic Mode Cancellation Pattern

Get the Fog Out's build menu (fig. 12c) includes an ease of use feature where closes automatically when the player clicks on a tile that is unusable, thereby exemplifying the automatic mode cancellation pattern. Most often this is a tile that is not powered, but whenever a lerp target is active during the tutorial, it also includes tiles outside the bounds of the lerp target. Most of this functionality is managed by other classes, however TutorialController contributes through its method TileAllowed(), which returns whether a selected tile can be built upon according to the criteria restricting building to within the lerp target. Besides providing an easy, conventional way to close the build menu, it also implies which tiles are available to build upon or not in case the lerp target wasn't clear enough.

Progress Pattern

The tutorial also exhibits the progress pattern by haivng a progress bar at the bottom of the objective window during the tutorial (fig. 13). At significant points in the tutorial, the bar's progress is incremented, giving the player an indication of how far along in the tutorial they are, and how much they have left to go. It was added after feedback that often players were waiting for the tutorial to finish, but had no idea how long they had left to complete it and were left wondering if they had to continue for another dozen stages or if they were near the end and about to begin playing the main game.

Restructure

As I mentioned earlier, there are a number of changes I would make, such as making TutorialController.ButtonsNormallyAllowed() a member of TileData as it more naturally belongs there and could from there be tweaked to be more efficient, and to implement a proper observer pattern between btnTutorial and TutorialController to improve efficiency. There are also various enum values and field and method names whose names are out of date compared to what they are being called from the player's perspective in the current game; those I would update to reflect their current names, the fields and enum values a few at a time to make sure I remember to reconnect anything that breaks.

More substantially, TutorialController is unnecessarily long, which makes it harder to maintain, and TutorialController and ObjectiveController are separate classes, which has led to some duplication of functionality and unnecessary complexity. To address these issues, I would make the more extensive structural changes:

- Of combining DialogueBoxController, TutorialController and ObjectiveController into one StageController class that handles all the stages of the game. This would remove the unnecessary duplication and complexity of having fields and functionality split amongst and duplicated across its three precursors.
- Of separating out all of the stages into separate classes that all inherit from an abstract Stage class. Each Stage would have additional fields and methods beyond the common Stage members as required to do their jobs.
- Of separating out other members that don't have anything to do with managing stages into other controller-type classes dedicated to those particular things. Identifiable across TutorialController and ObjectiveController are members that relate to in-world lerp target management, UI lerp target management, and monitoring of power and whether it's been overloaded. These members could be redistributed amongst Target, UILerpController, and PowerOverloadMonitor classes.

Splitting up existing classes, TutorialController in particular, amongst this greater number of interconnected Stage and controller-type classes would increase *Get the Fog Out*'s programmatic complexity in terms of the number of interlocking classes and pieces. However, it would make each of those pieces so much more manageable, as you would know: which piece to open up to affect what; where to find it to do so, rather than having to rifle through a lengthy, omniscient and omnipotent master class; and that each piece would only be given access to the pieces it required, reducing the confusion of which fields in a long list of fields were

used by which Stages of the game, and what other classes were interconnected to the StageController for what reason.

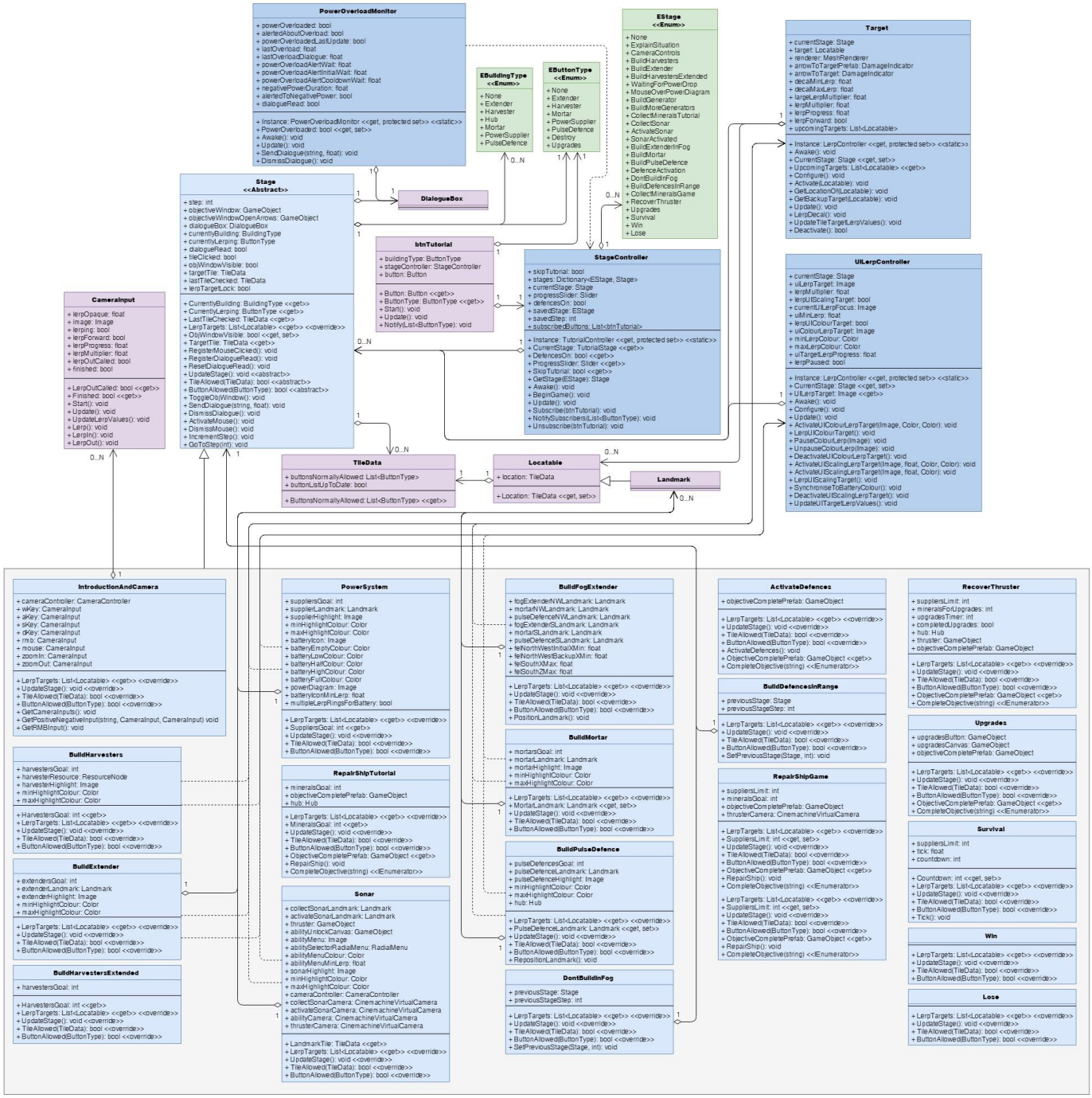

Figure 15: how I would restructure the objective related classes to be much shorter, more manageable, and more maintainable. Note: TileData is listed here as only having three members, but these are just the ones that would be redistributed to it upon a restructure; it has additional existing members irrelevant to the discussion of the objective classes that have not been listed for simplicity's sake.

Dialogue System

Data Structures

ColourTag

When text appeared in the dialogue box, we wanted it to scroll onto the screen several characters at a time. We also wanted it to be able to display text in different colours. When using HTML-like tags to achieve this, we found that part of the tag would be printed, before vanishing and then changing the colour of the text. To work around this, I created the ColourTag class to encapsulate single-character opening and closing tags for changing text to specified colours without having HTML printed onto the screen. It needed to have its values settable in the inspector and then readable by DialogueBox, but not later rewritable.

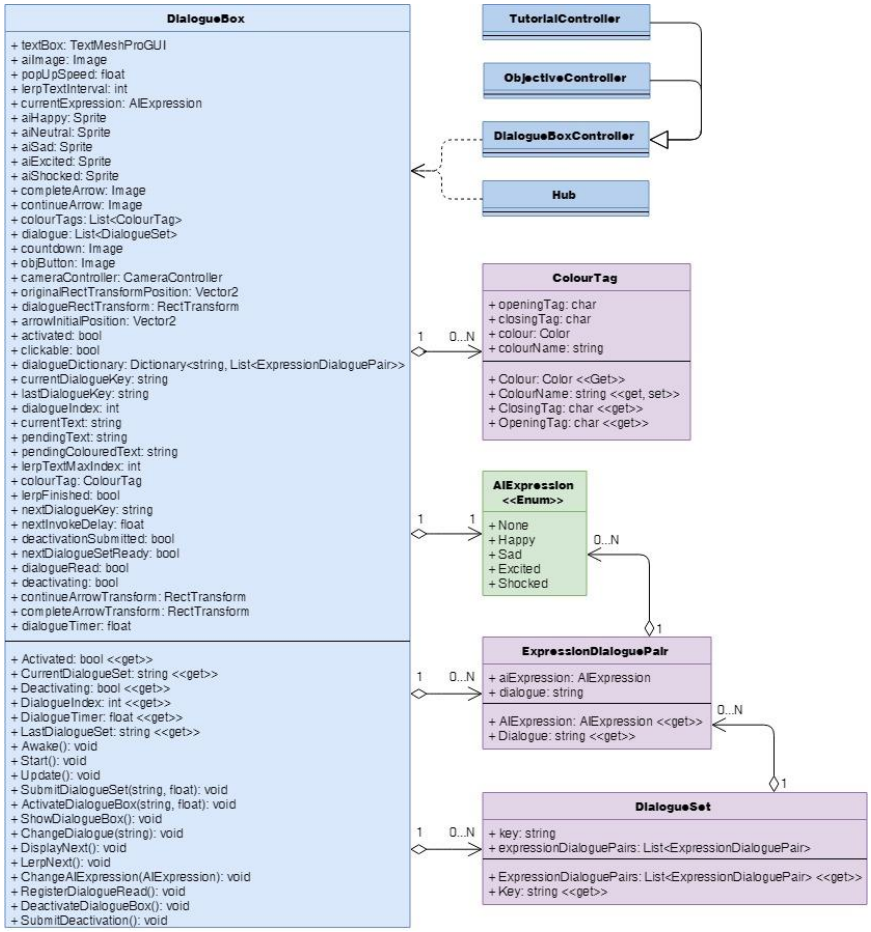

Figure 16: the current structure of DialogueBox and its related classes.

This could have been achieved with a class or struct; I chose to make it a class out of habit. A future iteration might consider converting it from a class to a struct if it being a struct would help the game run faster or consume less memory, but this would need to be tested. Similarly, a future iteration might consider doing away with using public properties by making the fields that never have their values changed (openingTag, closingTag and colour) public but read-only, such that they could be accessed directly rather than via their corresponding public property, thereby decreasing the number of steps taken to execute the code. I would test the performance gains here as well, but they seem more promising than the conversion from class to struct. The impact these two changes would have on performance would be negligible, as there would only be a handful of ColourTag objects stored in DialogueBox regardless, but they would represent better programming practice nonetheless.

The ColourName public property breaks the pattern of being a get-only property, having a setter accessed during DialogueBox.Awake(). Looking at it now, that could be made redundant and made into a get-only public property by giving ColourTag an Awake() method that does the setting of colourName rather than DialogueBox.Awake(). If other public properties were made read-only public fields, this one would be left as is, as it would need to be set in Awake() rather than in the inspector.

ColourTags: List

DialogueBox stores its ColourTags in a List, which is visible in the inspector. A List was chosen for this as I knew it would be visible in the inspector, and elements could be added or removed as necessary.

Looking through the code, once the List is defined, it never changes; the code either goes through all List elements or accesses specific elements. Consequently, an argument could be made for swapping the List for an array, which doesn't have the extra functionality of a List and therefore logically should be smaller in memory. Whether or not a future iteration converts to using an array would have to be informed by performance tests to assess changes in access speed and memory occupied, and double checking that arrays can be viewed in the inspector.

ExpressionDialoguePair

We wanted the on-screen dialogue box to display lines of dialogue and matching facial expressions of the AI assisting the player. I wanted them to be grouped together in the inspector, and locked 1:1. Rather than have a List of strings for dialogue and a List of AI expressions and have to track if they were the same length, I put together the ExpressionDialoguePair class to store each line with its matching expression. The only changes that might be worth making would be to make the fields public read-only, remove the public get-only properties, and make the class a struct, following the reasoning discussed regarding the same changes for ColourTag. Again, each of these changes would need to be tested for their performance benefit.

DialogueSet

When the on-screen dialogue box presented dialogue to the player, we wanted players to be able to go through it a chunk at a time rather than facing a wall of text, but for that whole block of text to still be submitted for displaying all at once rather than small chunk by small chunk. Consequently, I put together the DialogueSet class, which stores a string identifier and a List of ExpressionDialoguePairs that represent one or two lines of dialogue. Ideally, this information would have been represented in the inspector using a dictionary, with strings as the keys and Lists of ExpressionDialoguePairs as the values. However, Unity doesn't display Dictionaries in the inspector, which is where the string identifiers and ExpressionDialoguePairs get set. Therefore, this workaround had to be implemented to create Dictionary-like representation suitable for setting in the inspector.

Again, there are cases to be made for making the class into a struct, the public get-only properties into public read-only fields, and the List of ExpressionDialoguePairs into an array, cases that would need to be supported by test results.

Dialogue: List, Dictionary

Since we wanted to display many batches of dialogue to the player at different points in the game, we needed to be able to define multiple DialogueSets in the inspector, necessitating they be stored in a List exposed to the inspector. (Yes, again, List vs array testing could be justified as the List doesn't change once defined.) As discussed just above, I didn't make it a Dictionary of string-ExpressionDialoguePair key-value pairs since it needed to be visible in the inspector. However, for use once the program was running and the DialogueSets were defined, I did find that using Dictionaries was easier, as I could just use the string key to directly access the associated ExpressionDialoguePairs rather than have to loop through a List and manually check each string key, which would have resulted in longer and slightly less maintainable code. As such, I kept the List for setting values in the inspector, but in the Awake() method added a loop to take each DialogueSet's string and List of ExpressionDialoguePairs and create Dictionary entries where they were the keys and values respectively, and had the code after that point access dialogue content from the Dictionary rather than the List. The only change I would make here would be to use the Dictionary right from the start and do away with the DialogueSet class to remove code and thereby shorten and simplify DialogueBox.Awake() and DialogueBox.cs, but that is a non-viable change to make until Unity allows Dictionaries in the inspectors like Lists or I learn of a way to work around or bypass that limitation.

Design Patterns

State Pattern

The on-screen dialogue box has a number of identifiable states that it transitions between, with the state of being active having several "sub-states" of things it might be doing (fig. 17). Programmatically, in terms of DialogueBox's fields and the structure of its code, it does not look like it implements the state pattern; rather, it uses a set of if statements to check if this condition is true or that thing is ready to be done (fig. 18).

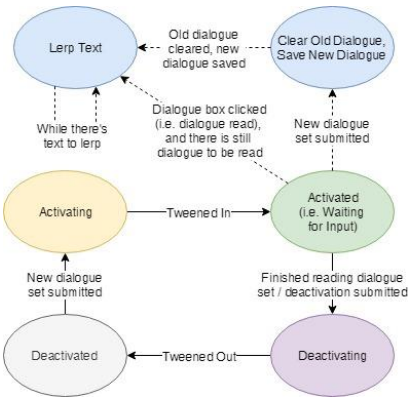

Figure 17: the identifiable states the dialogue box goes through.

This arguably makes it a lot more confusing and a lot less readable, even to the one who made it this way in the first place.

If how DialogueBox handles what it should be doing were to be restructured, I would code it to have a more formal, switch-based implementation of the state pattern similar to what TutorialController and ObjectiveController currently operate on, having cases for each of an enum's value that redirect Update() to state-specific methods that execute that state's behaviours and check the conditions to changing to a new state. This might lengthen DialogueBox's code, but it would clean it up and simplify it, making identification of its current state and maintenance regarding those states easier; there wouldn't be multiple checks for "Is it in this state?", "Is it in this other state?" and "Or this one?", but a single "Which state is it in?" "This one" "Do A, B and C" check. Such an implementation might also have some performance gains, as only what's relevant for the current state would get checked for execution, as opposed to the current implementation where currently everything has to be checked at least partially to verify if it should be executed or not. Any such gains would need testing to verify, but would regardless be secondary to the organisation and readability gains. If a substantially greater number of stages were to be added to DialogueBox, I might instead consider a class-based implementation of the state pattern as I have discussed for StageController above, but with the current limited number of states of DialogueBox, such an implementation would arguably be overkill for now.

```
if (nextDialogueKey != "" && !deactivating)
{
    if (!activated)
    {
        ActivateDialogueBox(nextDialogueKey, nextInvokeDelay);

        nextDialogueKey = "";
        nextInvokeDelay = 0f;
    }
    else
    {
        ChangeDialogue(nextDialogueKey);

        nextDialogueKey = "";
        nextInvokeDelay = 0f;
    }
}
else if (deactivationSubmitted && activated)
{
    lastDialogueKey = currentDialogueKey;
    currentDialogueKey = "";
    clickable = false;
    DeactivateDialogueBox();
}

deactivationSubmitted = false;

if (nextDialogueSetReady && dialogueDictionary.ContainsKey(currentDialogueKey) && dialogueIndex < dialogueDictionary[currentDialogueKey].Count)
{
    DisplayNext();
    nextDialogueSetReady = false;
}
else
{
    if (nextDialogueSetReady && (dialogueDictionary.ContainsKey(currentDialogueKey) || dialogueIndex >= dialogueDictionary[currentDialogueKey].Count))
    {
        nextDialogueSetReady = false;

        if (dialogueDictionary.ContainsKey(currentDialogueKey))
        {
            Debug.Log($"Warning: nextDialogueSetReady was true, but dialogue key {currentDialogueKey} doesn't exist in dialogueDictionary.");
        }
        else
        {
            Debug.Log($"Warning: nextDialogueSetReady was true and dialogue key {currentDialogueKey} exists, but dialogueIndex {dialogueIndex} is an in
        }
    }

    if (clickable && dialogueRead)
    {
        RegisterDialogueRead();
    }
}
```

Figure 18: a section of DialogueBox.Update() that handles what "state" DialogueBox is informally in, and what DialogueBox should therefore be doing.

Usability Patterns

State Pattern

As mentioned above, the on-screen dialogue box has a number of visible states that it goes through (fig. 17): being not visible, tweening in, being visible with dialogue, and tweening out again. The state of being active and ready to be interacted with is indicated by the dialogue box's visibility, motionlessness, and yoyo-ing arrows. When it tweens in or out, or simply isn't visible, that conveys to the player that it isn't available to be interacted with or at all. These conveyances indicate to the player what they can and can't do.

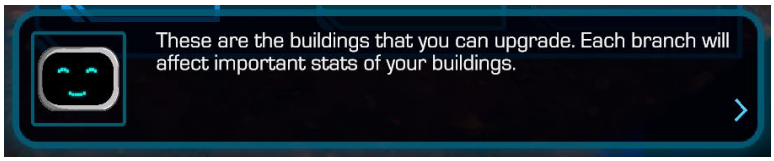

Figure 19: a dialogue box with a right-facing arrow, indicating that there is more dialogue to be read.

Progress Pattern

As noted just above, the dialogue box features yoyo-ing arrows that help indicate the dialogue box's intractability. Their direction also provides some indication of the player's progress through the dialogue set: when

there are still more lines of dialogue to be displayed, they'll point to the right and yoyo horizontally (fig. 19), but when the displayed dialogue is the last line for that dialogue set, they'll point down and yoyo vertically (fig. 10), indicating that the next click will result in the dialogue box tweening out. This doesn't convey the same level of detail as the tutorial's progress bar, but provides some conveyance of progress nonetheless.

Restructure

As noted in my discussion of data structures and patterns used in DialogueBox, there are a number of relatively minor adjustments that I would make in future iterations, including:

- Adding to ColourTags an Awake() method that sets its colourName field, so that the ColourName property could be made a get-only property.
- Testing if making ColourTag, ExpressionDialoguePair and DialogueSet structs rather than classes would be beneficial performance-wise, keeping the changes if they were, but otherwise reverting them.
- Making the fields of those classes that only get set in the inspector public read-only fields, doing away with their public properties, and double checking if that helps or hurts DialogueBox performance-wise.
- Testing if exchanging the Lists in those classes for arrays would affect performance, keeping them as arrays if it helped and reverting back to Lists if it didn't.

More major changes that I would make to the structure of DialogueBox.Update() include:

- Splitting the contents of Update() into different methods that have distinct purposes and are called from Update().
- Reorganising the transitioning between the current informal "states" to use a proper switch-based state pattern, with cases for values of an enum each redirecting to methods for handling execution during that state and checking the conditions for switching to a different state.

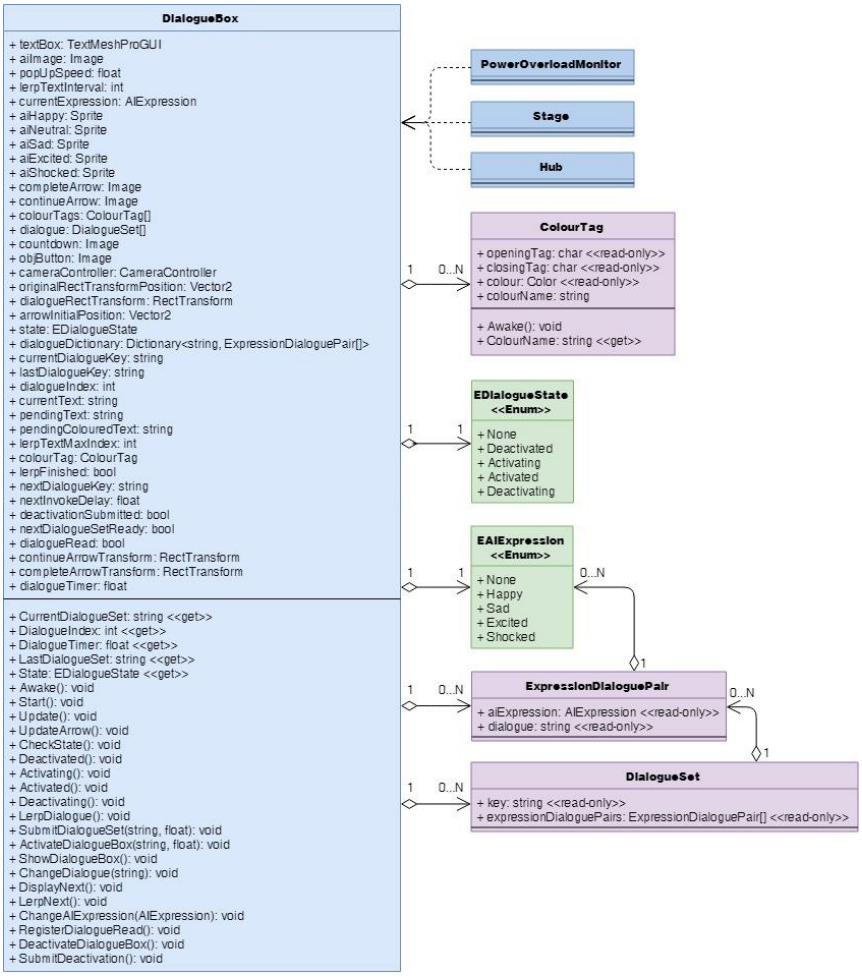

Figure 20: how I would restructure DialogueBox and its related classes to be more efficient and maintainable.

Restructuring DialogueBox.Update() in this way would improve readability and maintainability, making the code cleaner and easier to understand and manage.

Summary

The data structures used in each class were chosen for their impact on flexibility and performance, and for their usability with Unity where necessary. The data patterns used were chosen for their suitability for the particular problems of each class. Some choices and implementations are fine as is, while others leave room for improvement. The longer classes each have ways that they could be rearranged or broken up into smaller pieces to be more readable and manageable, and therefore easier to maintain, as discussed in each section and visualised in the UML diagrams of the restructuring of each set of classes. Just as the design of *Get the Fog Out* could be iterated upon and improved, so too could its internal workings to make it more readable and more performant.