Distinction Report Key Points

Fog

Data Structures

FogUnits: List, 2D Array

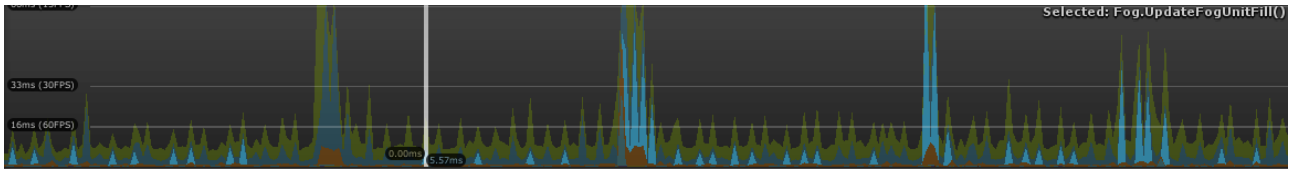

Figure 1: Fog.UpdateFogUnitFill() under the profiler while all FogUnits were being updated every time the method was called.

- I originally used a pooling system with 2601 FogUnits split between two lists and allocated to TileDatas at runtime from the pool. It wasn't very fast (fig. 1).
- Running "update" methods less than once per frame helped, but still left noticeable split-second pauses, particularly after the map went from 51x51 to 71x71 (with 5041 FogUnits).
- I tried the combination of: storing FogUnits in a 2D array, with their positions matched to their TileData's in their 2D array and on the map, for accessing when the FogUnit's position is known; keeping the in-play list for for and foreach loop processing but scrapping the pooling list; and spreading FogUnits' "updating" over all frames in the update interval rather than doing them all in a single frame. This resulted in a smoother framerate, with no noticeable performance drops attributable to the fog (fig. 2).

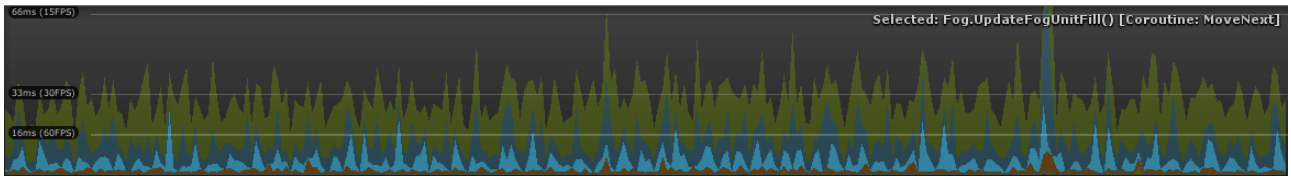

Figure 2: UpdateFogUnitFill() as an IEnumerator updating chunks of FogUnits each frame over the whole 0.25 second interval, rather than updating all of them within the one frame.

Fog Spheres: List

- FogSpheres used a two-list pooling system for their entire existence. When adding the FogUnits' 2D array, having one here wasn't appropriate considering FogSpheres weren't stationary and there were only a handful of them, making List.Contains()'s linear searches[1] for FogSpheres quick. 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

- FogSphere spawn points (i.e. FogSphereWaypoints with spawnPoint set to "true") were stored in a list.

[1] *List<T>.Contains(T) Method*, Microsoft.

- Their list is only used as a base list for picking random waypoints for FogSpheres to spawn at, being copied into a new list rather than be manipulated itself. Given that it never changes, it could be made into an array that lacks lists' extra functionality and should logically be smaller in memory, yet is as fast if not faster when accessed[2]. It would need to be implemented and tested against the current list's performance.

Design Patterns

State Pattern

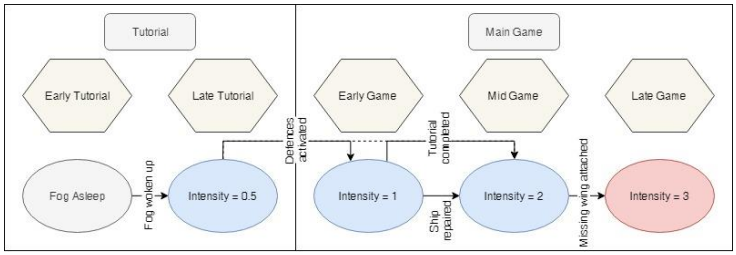

Figure 3: the informal "stages" of the fog.

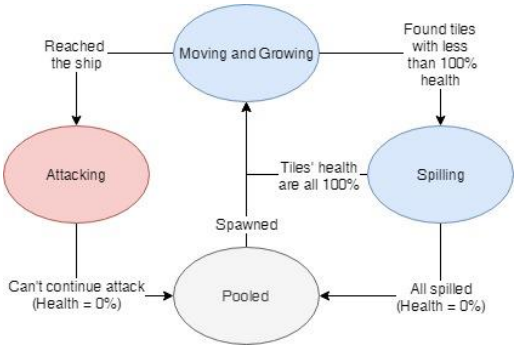

Figure 4: FogSphere's states.

- FogUnits included an informal implementation of the state pattern[3] (fig. 3), with Fog featuring an Intensity public property updating the fog's behavioural parameters. This worked simply and cleanly, and should be kept.
- More formally, FogSpheres have distinct behavioural states (fig. 4), indicated by the FogSphereState enum. Fog.UpdateFogSpheres() checks it and calls the appropriate state methods, executing their behaviour, including checking state change conditions. This worked okay, but does leave room for improvement (see below).

Singleton Pattern

- Fog, like other manager-type classes, implemented the singleton pattern[4] to allow universal access and ensure its uniqueness.
- Its uniqueness ensures a duplicate Fog with an extra 5041 FogUnits wouldn't be instantiated and kill *Get the Fog Out*'s (*GTFO*'s*)* performance. Therefore, I would not change this aspect.

Factory Pattern

- Fog includes factory pattern-like[5] methods for standardised creation of FogUnits and FogSpheres when the fog is initialised, and for instantiating and activating them when they are spawned.
- This helped keep Fog's code clean and maintainable, as spawning only required method calls, not blocks of code.
- The methods' inclusion in Fog rather than a dedicated FogFactory class resulted in Fog being longer and harder to search through, hindering maintainability.

Component Pattern

- Unity's GameObject class and its components implement the component pattern[6] [7]. *GTFO* uses GameObjects, thereby including the component pattern by default.

---

[2] *When should I use a List vs a LinkedList*, Allen 2008.
[3] *State Design Pattern*, SourceMaking.com.
[4] *Singleton Design Pattern*, SourceMaking.com.
[5] *Design Pattern – Factory Pattern*, Tutorials Point.
[6] *Component*, Robert Nystrom.
[7] *GameObject*, Unity Technologies 2019.

Usability Patterns

State / Progress Patterns

- The fog informally implements the state and / or progress usability patterns[8] thanks to its changes in colour through the game that match the intensity and aggressiveness of its behaviour, from a black-ish grey (asleep) to blue (awake) to red (the final stage of the game).

Restructure

- To restructure the fog-related classes:
    - I would remove methods (e.g. FogSphere's UpdateDamageToFogSphere(), DealDamageToFogSphere()) and enumeration values with related functionality (e.g. FogExpansionDirection.Orthogonal and its corresponding code in Fog.ExpandFog()) that were obsolesced by design decisions and won't be used again, which would result in (negligible) performance gains and shorter, more manageable, more maintainable code.
    - Fog is quite long, so I would break it up into manager classes for FogUnits, FogSpheres and FogLightning, and a factory class that could create each of those classes for their respective managers. FogFactory and each manager class would be singletons to ensure uniqueness and universal access.
    - I would dismantle FogUnit's inheritance structure and use those separate classes as components of FogUnits. This would make Entity a Health component that FogSphere could also outsource code to, it's health functionality being near identical to FogUnit's. FogUnit and FogSphere's common rendering members could also be made into a separate shared component class.
    - I would amend *GTFO*'s current implementation of the state pattern and split the methods associated with each of FogSphere's states into separate classes that implement a base State class and operate on FogSpheres.
- This would create a more intricate network of which classes use each other, but each would be more singular in their purpose, shorter, more readable and maintainable, and more easily debugged, and any performance losses from increased public properties would be infinitesimal[9] [10].
- For further iterations, I would want to look into the FogEntityRenderer and Health components using the flyweight pattern of splitting classes into shared and unique functionality[11], as a lot of their information is identical across all FogUnits and FogSpheres.

Objectives (Tutorial / Main Game)

Data Structures

Lerp Targets Remaining: List

- TutorialController keeps a list of all Locatables to which upcoming tutorial stages will move the lerp target guiding players. The list affords painless appendment / removal. Any speed loss from not using an array (which is faster at searching) is minimised by the small number of Locatables and outweighed by the appendment / removal performance gains**Error! Bookmark not defined.**.

[8] *Usability Patterns in Games*, Woodard 2019.

[9] *Performance overhead for properties in .NET*, Apoorv020 2010.

[10] *Field vs Property. Optimisation of performance*, Boppity Bop 2012.

[11] *Flyweight Design Pattern*, SourceMaking.com.

ButtonsNormallyAllowed: List

- ButtonsNormallyAllowed() compiles the list of buttons normally available on a given tile. The available buttons can vary, so the list needs to be built from scratch when needed, making lists, which are good with appendment, more suitable than fixed-size arrays that would waste space**Error! Bookmark not defined.**.
- One could argue ButtonsNormallyAllowed() should be a member of TileData, such that anything that needs to check it can without going past TutorialController unless they need a list of what's currently permissible. Doing it this way, I'd have it check if the list needs to be compiled, compiling and storing it if so, and then return the list. (Dismantling a Harvester on a depleted ResourceNode makes its tile a normal one, so a bool flag requesting recompilation would have to be set.) This would improve performance as the lists would only be compiled twice at most, rather than every time they're accessed

Local Data Structures of Methods: List

- TutorialController.DefencesOperable(), PositionFogExtenderLandmark() and GetBackupTarget() also use lists, each compiling them to search or to compare with another list. Lists are most appropriate because they facilitate simple, flexible and efficient appendment, removal and searching**Error! Bookmark not defined.**.

Design Patterns

State Pattern

- *GTFO* implements the state pattern[3] for its tutorial and main-game objectives (fig. 5), with a switch checking an enum field and redirecting to methods for each stage, methods with a switch for substages, their functionality, and their conditions for progression.
- Switch-based stages were used because they were simple, straightforward and worked, but resulted in TutorialController and ObjectiveController being quite long and harder to maintain.
- Separating tutorial and main-game stages was an accidental by-product of how we were delegating work and who would do what, and led to unnecessary complexity and duplication.

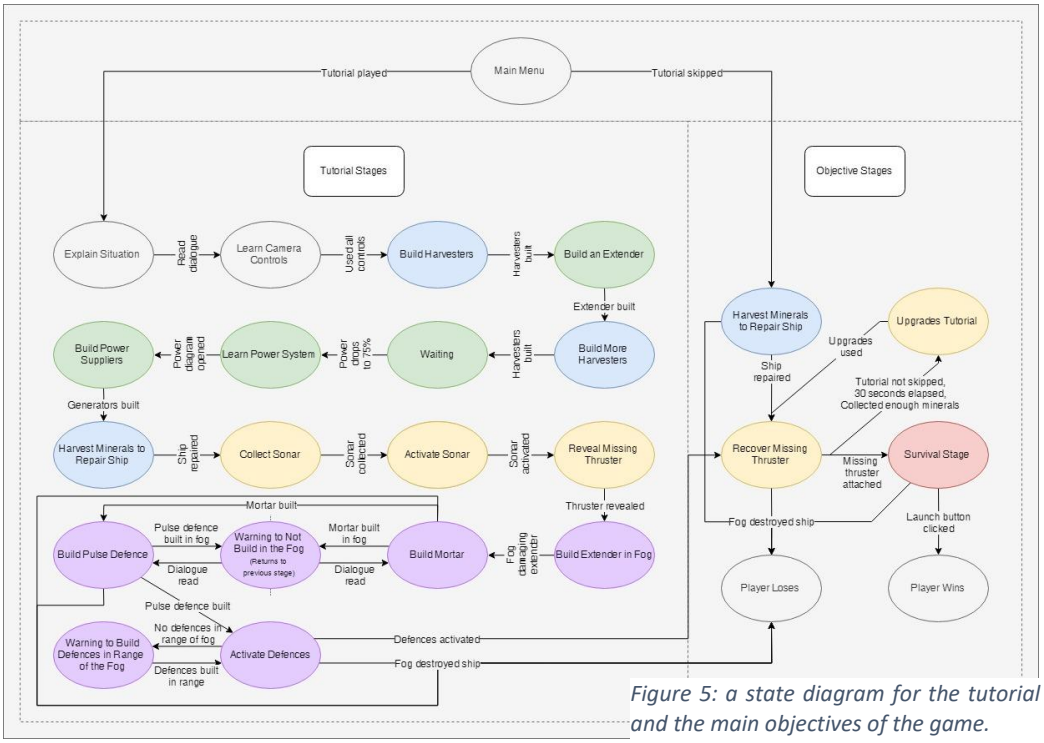

Figure 5: a state diagram for the tutorial and the main objectives of the game.

Singleton Pattern

- The stage controllers both implement the singleton pattern[4] to ensure uniqueness and universal access. They implement it separately rather than via DialogueBoxController to avoid any complications that might arise from their both being its children, and to avoid requiring casting every time they're accessed.

Observer Pattern

- btnTutorial and TutorialController interact inversely to the observer pattern[12], btnTutorial checking every update with TutorialController if it's button should be interactable, rather than subscribing to have TutorialController tell it when it should or shouldn't be interactable. This was used because it was simple and it worked, but it's less efficient than a proper observer-pattern implementation.

Usability Patterns

Focus Pattern

- The tutorial extensively uses the focus pattern**Error! Bookmark not defined.** to teach players *GTFO*, using lerp targets in the UI and on the map and interaction-appropriate diagrams to draw players' attention.
- The tutorial also restricts player input to what it wants players to do, forcing them to participate to progress, but ensuring they learn from experience rather than theory.

Magnetism Pattern

- The tutorial features several manifestations of the magnetism pattern**Error! Bookmark not defined.**:
    - Lerp targets getting players to build on specific tiles and UI lerp targets getting players to use particular buildings, so as to reinforce what can be built where and prevent illegal moves during the tutorial.
    - If a lerp target would be on an already occupied tile, it moves to a nearby tile that's not occupied or earmarked, preventing the tutorial requesting illegal moves of the player.
    - The lerp target for the extender in the fog being placed where a player is already extending around the rocks between the ship and the missing thruster, or where they can first extend out to the fog if they haven't done so yet. The defensive buildings have their lerp targets placed in the same area so that players expand outwards and start clearing fog in the same direction.

State Pattern

- The stage controllers feature several manifestations of the state usability pattern**Error! Bookmark not defined.** to indicate changes in options or status:
    - The fog changing colour as it becomes more aggressive, conveying its behavioural state.
    - btnTutorial making buttons opaque and interactable or faded and uninteractable as the current stage requires, matching their appearance to their intractability, and conveying what they should build during the tutorial.
    - The objective window tweens onto the screen with updated text when the objective changes, and pulsating buttons appear for players to complete major objectives. (The associated methods are UIController's members, but are called by the stage control-

[12] *Observer Pattern*, SourceMaking.com.

lers.)

Automatic Mode Cancellation Pattern

- The build menu closes automatically when players click on unusable tiles (whether from being unpowered or outside a lerp circle), providing automatic mode cancellation**Error! Bookmark not defined.** and conveying which tiles can be built upon.
- Most of this functionality is encapsulated in other classes, but TutorialController contributes via TileAllowed(), which informs MouseController if a tile is interactable.

Progress Pattern

- The progress pattern**Error! Bookmark not defined.** is exemplified by the progress bar in the objective window during the tutorial. It increments when key objectives are completed and conveys players' progress through the tutorial.

Restructure

- To restructure the tutorial classes, I would:
    - update the names of fields and enum values that are out of date compared to what various things are now called from the player's point of view, doing so incrementally so that anything that breaks is caught.
    - move ButtonsNormallyAllowed() to TileData and tweak it to maximise its efficiency.
    - implement a proper observer pattern between btnTutorial and TutorialController so that btnTutorial isn't hassling TutorialController every update.
    - combine DialogueBoxController, TutorialController and ObjectiveController, removing unnecessary duplication and complexity.
    - split stage methods into classes inheriting from a Stage base class and encapsulating the fields and methods they require during their part of the game.
    - Split members irrelevant to managing stages into other dedicated controller-type classes. Identifiable are members relating to in-world lerp target management, UI lerp target management, and monitoring power overloading.
- Breaking up existing classes would again increase the complexity of the network of what uses what, but would make each piece more manageable and focused on the objects and tasks allotted to it.

Dialogue System

Data Structures

ColourTag

- ColourTag encapsulates a colour, its name, and opening and closing tag characters for changing dialogue text's colour, as HTML-like tags got partially printed before rendering properly. It needed to be inspector-viewable and readable but not writable in-code.
- ColourTag seems appropriate to be made into a struct, which might make help performance marginally[13] (this would require verification testing).
- The public properties could be made into read-only public fields, but the performance gains would be negligible[10] and it would make ColourTag not viewable in the inspector[14]. Therefore, they should not change.
- Looking at the code, ColourTag.ColourName could be set in a ColourTag.Awake() method rather than DialogueBox's, or in the property itself before returning it when it's first accessed,

[13] *Choosing Between Class and Struct*, Microsoft 2008
[14] *Script Serialization*, Unity Technologies 2019

allowing ColourName to be made a get-only public property

ColourTags: List

- DialogueBox stores ColourTags in a list, which I knew to be inspector-viewable, and would allow appendment / removal as necessary.
- The code iterates through the list or views particular elements, but never changes it. It could be made an array instead, which is also inspector-viewable[14], should logically be smaller in memory, lacking lists' extra functionality, and be at least as fast for sequential / random access**Error! Bookmark not defined.**. (The latter two would require verification via testing.)

ExpressionDialoguePair

- We wanted the dialogue box to display dialogue and matching AI facial expressions, having them visible together in the inspector and matched 1:1. Rather than use two lists and track their lengths, I made ExpressionDialoguePair to store the two together.
- ExpressionDialoguePair could be made a struct for the same reasons as ColourTag, but performance gains would need verification via testing.

DialogueSet

- We wanted dialogue to be presentable one line at a time rather than as a wall of text, but for that whole block to be triggerable with one call. For this, I made DialogueSet to hold a list of ExpressionDialoguePairs and a string key.
- Again, DialogueSet could be made a struct and its list an array, but any gains would need to be tested for.
- I would have made DialogueSet's contents keys and values in a dictionary, but the values need to be set in the inspector and dictionaries aren't inspector-viewable, necessitating this as part of a workaround.

Dialogue: List, Dictionary

- Again, we wanted to set in the inspector many batches of dialogue for the player to encounter, and so made a list of DialogueSets. (Yes, again, List vs array testing could be justified as the List doesn't change once defined.)
- As noted above, dictionaries aren't inspector-viewable, so were out of a question for setting values. However, they were easier to use in the program once the values were set, requiring only providing a key to access values rather than loops checking for matching keys. Consequently, DialogueBox.Awake() converts the list of DialogueSets to a dictionary of string-ExpresisonDialoguePair key-value pairs, making the code slightly faster and shorter yet still inspector-viewable.

Design Patterns

State Pattern

- DialogueBox has several identifiable "states" that it changes between, the active "state" having several "substates" (fig. 6). Programmatically, it doesn't use the state pattern[3], but a set of less readable if statements all checking conditions for different behaviours.
- To clean that up, I would implement in DialogueBox a switch-based state pattern like the stage controllers currently use, with an enum field tracking the state and a switch redirecting to state-appropriate methods. This

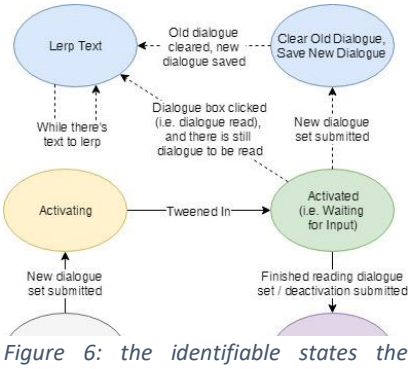

Figure 6: the identifiable states the dialogue box goes through.

might be faster (only checks relevant to the current state would be performed), but this would be secondary to the organisation and redability gains.
- If DialogueBox were to have more "states", a class-based state pattern implementation might be better, but for now that's overkill.

Usability Patterns

State Pattern

- As noted above, the dialogue box goes through a number of visible states (fig. 6): being not visible, tweening in, being visible with dialogue, and tweening out again.
- Interactability is indicated by the dialogue box being visible and stationary with yoyo-ing arrows; uninteractability is indicated by motion or lack of visibility

Progress Pattern

- The dialogue box's yoyo-ing arrows (noted above) also indicate players' progress through a dialogue set. On the last line of dialogue, they point down and yoyo vertically, suggesting it'll tween out. Otherwise, they point right and yoyo horizontally, conveying "continue reading". This isn't as much detail as a progress bar, but provides information nonetheless.

Restructure

- To restructure the dialogue box classes, I would:
    - remove ColourTag.ColourName's setter block by configuring ColourTags.Awake() or ColourName's get block to set colourName before retrieval, testing which one is more efficient.
    - test if making ColourTag, ExpressionDialoguePair and DialogueSet structs rather than classes would benefit performance.
    - swap their lists for arrays and test any changes in performance.
    - split DialogueBox.Update() into more focused methods, calling them from Update().
    - implement a proper switch-based state pattern, with cases for values of an enum redirecting to state methods.
- Restructuring DialogueBox.Update() like this would improve readability and maintainability, making the code cleaner and easier to understand and manage.

Bibliography

Allen, J 2008, *When should I use a List vs a LinkedList*, Stack Exchange, viewed 15 November 2019, <https://stackoverflow.com/questions/169973/when-should-i-use-a-list-vs-a-linkedlist/29263914#29263914>.

Apoorv020 2010, *Performance overhead for properties in .NET*, Stack Exchange, viewed 14 November 2019, <https://stackoverflow.com/questions/3264833/performance-overhead-for-properties-in-net>.

Boppity Bop 2012, *Field vs Property. Optimisation of performance*, Stack Exchange, viewed 14 November 2019, <https://stackoverflow.com/questions/9842917/field-vs-property-optimisation-of-performance>.

Microsoft, *List<T>.Contains(T) Method*, Microsoft, viewed 14 November 2019, <https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.contains?redirectedfrom=MSDN&view=netframework-

4.8#System_Collections_Generic_List_1_Contains__0_>.

Microsoft 2008, *Choosing Between Class and Struct*, Microsoft, viewed 15 November 2019, <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/choosing-between-class-and-struct>.

Nystrom, R, *Component*, GameProgrammingPatterns.com, viewed 15 November 2019, <http://gameprogrammingpatterns.com/component.html>.

SourceMaking.com, *Flyweight Design Pattern*, SourceMaking.com, viewed 15 November 2019, <https://sourcemaking.com/design_patterns/flyweight>.

SourceMaking.com, *Object Pool Design Pattern*, SourceMaking.com, viewed 15 November 2019, <https://sourcemaking.com/design_patterns/object_pool>.

SourceMaking.com, *Singleton Design Pattern*, SourceMaking.com, viewed 15 November 2019, <https://sourcemaking.com/design_patterns/singleton>.

SoruceMaking.com, *State Design Pattern*, SourceMaking.com, viewed 15 November 2019, <https://sourcemaking.com/design_patterns/state>.

Tutorials Point, *Design Pattern – Factory Pattern*, Tutorials Point, viewed 15 November 2019, <https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>.

Unity Technologies 2019, *GameObject*, Unity Technologies, viewed 14 November 2019, <https://docs.unity3d.com/ScriptReference/GameObject.html>.

Unity Technologies 2019, *Script Serialization*, Unity Technologies, viewed 16 November 2019, <https://docs.unity3d.com/Manual/script-Serialization.html>.

Woodard, C 2019, “Usability Patterns for Games”, *COS30031 Games Programming*, Learning Materials on Canvas, Swinburne University of Technology, 9 September, viewed 9 September 2019.