Sprint 2 -- 2 Week Duration (Ends 17th of March, 2017) 40-60 hours of work

Basic Design

The active design is an implementation of Paper Prototype 2.2, which specifies its mechanical requirements. Notions of story or aesthetics are deemed a distraction, and the technology for the prototype remains Unity.

Risk List

1. It might be tedious for designers to add new levels of complexity or define new complexity ladders.
2. I might not have a working prototype by the end of the sprint.
3. The coupling between UI and game logic might make both implementations brittle and susceptible to change.
4. I might spend too much time over-architecting things.
5. The complexity of my dependency graph might make it very difficult to reason about what's going wrong and what I need to do.
6. Controlling blob movements in real-time might severely break the logic of Paper Prototype 2.2.
7. Controlling blob movements in real-time might be more challenging to reason about than doing so in discrete time, since players might need to think on their feet more.
8. I don't know what exactly MapNode and MapGraph should have within them.
9. I don't know how tubes, highways, societies, and resource depots are going to interrelate at a programmatic level.
10. I don't know the precise relationship between blob sites, societies, and map nodes.
11. I don't know how construction and highway upgrading is going to function.
12. Coupling between various components might make them too interdependent.
13. Decoupling between various components might make the interstitial code too complex to reason about easily.
14. SimulationControl and UIControl might see their size balloon out of control.
15. I might need to pass map configurations through Unity's serializer, which could lead to all sorts of problems.
16. I might spend so much time designing "proper" code that I fail to implement usable behaviour.
17. I might need to manipulate the contents of BlobSites in the editor, and thus subject them to editor scripts and serialization.

Risk Mitigation

1. It might be tedious for designers to add new levels of complexity or define new complexity ladders.
   a) ComplexityLadders and ComplexityDefinitions should be established as either MonoBehaviours or as ScriptableObjects, so that they can be manipulated in the inspector. Ascent costs and descent timings should be defined inside of ComplexityDefinition as a field. For now, since there's only one way of getting from a given complexity to another, all ascensions to a given complexity should cost the same.
2. I might not have a working prototype by the end of the sprint.
   a) I must prioritize the completion of meaningful units of behaviour (societies, tubes, highways, etc) over the specific ways in which they are integrated, making sure to decouple

modules as much as I can. Only once I have working implementations for blob sites, societies, tubes, highways, and resource depots should I focus on integrating these various pieces into the behaviour I want.

3. The coupling between UI and game logic might make both implementations brittle and susceptible to change.
    a) I will keep the SimulationControl and UIControl modules as the primary ways in which UI interacts with game logic and vice-versa. I'll maintain the separation between a given interface for a class and the set of data that the UI needs to know about, and maintain the separation between UI commands and the code that implements it. That way, the structure of the game code can change drastically and independently of the UI code, and vice versa.
4. I might spend too much time over-architecting things.
    a) We'll start with decoupled modules that'll be thoroughly unit-tested. Then we'll figure out some (probably messy at first) paradigm for linking all of the pieces together. All major architectural decisions must be made up front, and we cannot make any major changes to the overall structure of the codebase until the next sprint. Beyond that, we must focus on the product we are delivering, not the prettiness of the code that gets us there, especially for the prototype.
5. The complexity of my dependency graph might make it very difficult to reason about what's going wrong and what I need to do.
    a) Decoupling, each module must manage a single task, implement in as much isolation as possible.
6. Controlling blob movements in real-time might severely break the logic of Paper Prototype 2.2.
    a) This is one of the things we're trying to test with Digital Prototype 2.2. This prototype is the risk mitigation for that concern. Building it quickly and playing with it early are paramount.
7. Controlling blob movements in real-time might be more challenging to reason about than doing so in discrete time, since players might need to think on their feet more.
    a) Another thing Digital Prototype 2.2 is trying to test.
8. I don't know what exactly MapNode and MapGraph should have within them.
    a) Why do we need MapNodes in the first place?
        - To specify places that various map features might be created.
    b) Does the MapNode care about the map feature upon it?
        - No. The MapNode only cares about its relationship to nodes and edges in its graph.
    c) Does the feature upon a MapNode care about the MapNode it's on?
        - To some degree. Since all tubes connect between MapNodes, a given feature needs to know its position to understand what tubes will be feeding it.
    d) But what do blob-receiving and distributing features actually care about?
        - Having blobs placed into them, having blobs removed from them.
    e) I think the only meaningful connection between a structure like a Society and a MapNode is through a given BlobSite. Thus I think it prudent to give every MapNode a BlobSite, then point Societies and ResourceDepots and whatnot at the MapNode they're on. They can then access the BlobSite through the MapNode, which will then allow the particular feature on a MapNode to change without fucking up tube connections. BlobTubes shouldn't give a damn about MapNodes, instead focusing on BlobSites.
    f) Map features might provide specific Alignment strategies to the BlobSites they're associated with, or blob alignment might be independent of them. That's more of a UI concern than a mechanical one.
9. I don't know how tubes, highways, societies, and resource depots are going to interrelate at a programmatic level.

    a) Item 8 contains some ideas on that. Highways probably need to possess tubes. Tubes need to interact with BlobSites directly. ResourceDepots do little more than change the capacities of the BlobSites they associate with. Societies should gain resources by listening to events on their BlobSites. Only the distribution of blobs into BlobHighways is complex, but the logic there is sufficiently complex that it must be managed by ConnectionCanon, or some similar task, since the only way of determining which blob goes down which tube is to check many highways. It makes sense to separate that more-global logic into its own class.

10. I don't know the precise relationship between BlobSites, Societies, and MapNodes.
    a) Already addressed in mitigation 8.
11. I don't know how construction and highway upgrading is going to function.
    a) It's most likely an integration issue. First, I need to get highways and BlobTubes to work at all. With the notion of BlobSites that we have now, BuildingConstruction is almost entirely an interstitial issue. Set the capacity on the underlying BlobSite, wait until that capacity is reached, and then trigger. Tube construction and upgraders will be a little different, as they need to intercept blob insertions into the tube they're trying to build/upgrade. That can most likely be tied in at the ConnectionCanon.
    b) The only issue is how to set the capacity of a MapNode's BlobSite. How do we switch between type-constrained and ambivalent BlobSites? What we can do is combine the two implementations. BlobSite maintains per-resource capacities as well as a global capacity. Structures set those capacities as needed. Construction and upgrade sites subscribe to the BlobSite's various events to check whether or not their completion conditions are met.
12. Coupling between various components might make them too interdependent.
    a) We've already done a lot of work in the risk mitigation addressing coupling. Especially if we unit-test a lot of the modules independently, there shouldn't be a coupling problem. Our main issue up to this point has been destructive redesigns, not spaghetti code.
13. Decoupling between various components might make the interstitial code too complex to reason about easily.
    a) We should be able to mitigate this risk just by making sure that each interstitial connection has a clear name and a single well-defined purpose, self-contained as much as possible, and relying on as few other interstitial modules as possible, ideally none at all.
14. SimulationControl and UIControl might see their size balloon out of control.
    a) For SimulationControl, we can address this with sensible componenting as it arises, by defining strategies for addressing each individual command and making classes that address logically similar collections of these commands. We can address UIControl bloating by using Chain of Responsibility intelligently, making UIControl primarily a delegator to other places where the logic of the UI is actually carried out.
15. I might need to pass map configurations through Unity's serializer, which could lead to all sorts of problems.
    a) If I define every class that might need to be serialized as either a MonoBehaviour, a ScriptableObject, or a struct containing only primitive data types, then I should be able to bypass the weaknesses of Unity's serializer. That will require avoiding dictionaries, however, which might be troublesome.
    b) We might need to find or create a structure that serializes and deserializes dictionaries into a format that Unity can make sense of.
16. I might spend so much time designing "proper" code that I fail to implement usable behaviour.
    a) This is a significant danger. I'll need to keep in mind that I'm building this code for a reason. I think reducing it to more manageable chunks will help keep me on track. And enforcing a no-rearchitecting rule will keep me from senselessly clobbering over existing work. Spotless code is not necessary here.

17. I might need to manipulate the contents of BlobSites in the editor, and thus subject them to editor scripts and serialization.
    a) I can manage that by making the BlobSites MonoBehaviours (which they'll probably already need to have ConnectionPoints for the various connecting BlobHighways) and then serializing their contents. I might consider making BlobPiles serializable in this way, or I might simply remove the notion of BlobPiles and incorporate much of their code into BlobSites. If BlobTubes are holding onto their contents sensibly, I might not need reservation code anymore, and instead let ResourceBlobs that are inappropriate for a given BlobSite sit in the tube, or be destroyed, or whatnot. We'll see about simplifying it.
    b) I might also be able to use int casting of the ResourceType enum to simulation Dictionary<ResourceType, X> by simply indexing into a List<X> with the int value of the desired ResourceType.

Sprint Backlog

To Do [Estimated 17 1/2 hours of work remaining] **17 1/2 hours of work incomplete by end of sprint, only 8 1/2 of which was considered critical**

1. Add unit-tested confirmation to the behaviour of BlobHighway endpoint-based content collections (1 hour).
2. Fill in the ResourceDepotFactory unit tests (30 minutes).
3. Figure out how to properly create ConstructionZones in the editor (3 hours).
4. Figure out how to properly create Highways in the editor (3 hours).
5. Figure out how to properly create ResourceDepots in the editor (3 hours).

In Progress

1. Create and debug the full integration until it's in working order (3 hours).

Completed

1. Separate components into sensible namespaces to more easily enforce decoupling (1 hour).
   a) Took ~ 1 hour to complete, though the boundaries for the task were fuzzy and the decision to call it complete was somewhat arbitrary.
2. Modify the interface for BlobSite to handle its new needs (1 hour).
   a) Took 1 1/2 hours, but involved the performance of a lot of other nonspecific tasks in order to enable compilation.
3. Build unit tests that prove a BlobSite implementation (3 hours).
   a) Took 3 1/2 hours.
4. Greenlight the BlobSite unit tests (3 hours).
   a) Took 1 hour.
5. Modify the MapNode module to include a factory, and provide a BlobSite for every MapNode and MapEdge (1 hour).
   a) Took 1 hour.
6. Modify the SocietyBase interface to handle its new needs (1 hour).
   a) Took 30 minutes.
7. Repair the Society unit tests to prove the new Society implementation (1 hour).
   a) Took 1 1/2 hours.
8. Greenlight the Society unit tests (1 hour).
   a) Took 2 1/2 - 3 hours (there was a serious Pomodoro interruption and I couldn't record the time properly).
9. Repair the BlobTube unit tests (1 hour).
   a) Took 30 minutes.
10. Greenlight the new BlobTube unit tests (1 hour).
    a) Was completed by executing the above task; took no additional time.
11. Augment the BlobTube unit tests with the requirements of BlobHighway, and then greenlight them (2 hours).
    a) Took ~ 45 minutes.
12. Define the BlobHighwayBase interface and construct unit tests to prove its implementation (2 hours).
    a) Took ~ 2 1/2 hours (interrupted by prerequisite tasks).

13. Greenlight the BlobHighway unit tests (1 hour).
    a) Took between 1 and 1 1/2 hours.
14. Define the ResourceDepotBase interface, then create unit tests to prove its implementation (1 hour).
    a) Took ~ 45 minutes.
15. Greenlight the ResourceDepot unit tests (1 hour).
    a) Took ~ 25 minutes.
16. Create an inspector-friendly implementation of ComplexityDefinition and ComplexityLadder. (1 hour)
    a) It was done by the time I got to it. Not sure how long it took.
17. Define the ConstructionZone interface and construct unit tests to prove its implementation (2 hours).
    a) Took ~ 2 hours.
18. Greenlight the ConstructionZone unit tests (1 hour).
    a) Took ~1 hour.
19. Define the HighwayUpgrader interface and construct unit tests to prove its implementation (2 hours).
    a) Took 2 hours.
20. Greenlight the HighwayUpgrader unit tests (2 hours).
    a) Took ~20 minutes.
21. Define the needed commands within SimulationControl (1 hour).
    a) Took 30 minutes.
22. Add the methods missing from the various modules that SimulationControl touches, along with their corresponding unit tests (3 hours)
    a) Took between 5 1/2 and 6 hours. There were a lot of things that needed fixing, and several suites of unit tests were missing key functionality (neither Tube nor Society even had factory implementations). Considerable unforeseen consequences.
23. Create a set of integration tests that test a SimulationControl implementation (5 hours).
    a) Took ~ 5 hours, interjected by the above task (which is likely related).
24. Greenlight the SimulationControl integration tests (3 hours).
    a) Took between 1 1/2 and 2 hours. Its completion produced another, smaller task.
25. Build the FactoryDependency hierarchy in the inspector (1 hour).
    a) Spent about 30 minutes on. The boundaries and intents of the task weren't clear.
26. Figure out how to more effectively break down the tasks remaining in the sprint backlog (30 minutes).
    a) Took ~20 minutes.
27. Design a UI that allows the manipulation of Highway permissions and priorities in-game (3 hours).
    a) Took ~3 hours.
28. Design a UI that allows for the construction of highways in-game (3 hours).
    a) Took ~2 1/2 hours.
29. Design a UI that allows the creation of ConstructionZones in game (3 hours).
    a) Took ~1 hour 20 minutes, but does not include the faculties for destroying construction zones.
30. Design a UI that allows for the destruction of ConstructionZones (2 hours).
    a) Took ~1 hour.
31. Design a system that allows for the creation of various societies in the editor (3 hours).
    a) Took ~ 1 1/2 hours.
32. Design a UI that allows for the upgrading of Highways in the game (2 hours).

    a) Took ~ 1 1/2 hours.
33. Wire all the UI-able modules to accept various EventSystem events and push them into UIControl (1 hour).
    a) Took ~45 minutes.
34. Construct the interface for BlobDistributor, and a set of tests that prove its implementation (2 hours).
    a) Took ~2 1/2 hours. **Completed after the end of the sprint.**
35. Greenlight the BlobDistributor unit tests (2 hours).
    a) Took ~2 hours. **Completed after the end of the sprint.**

Abandoned

1. Repair the BlobHighway unit tests (2 hours).
    a) Declared as contextually irrelevant given that the interface, BlobHighwayBase, needs to be properly defined, and thus the unit tests need to be rewritten, not merely repaired.

Sprint Review

It's hard to say the precise place that a review holds holds in this version of Scrum regardless, considering that I'm the only stakeholder. But for this particular sprint, it's a fairly easy consideration. While I created the majority of the pieces and integrations I needed to achieve digital prototype 2.2, I did not end with a playable product. I did not finish the last few critical integration tasks.

Sprint Retrospective

What went well?

- I managed to fairly consistently hit the six-hour days and the 60-hour work duration that I was aiming for. There was only one day during the duration of the sprint that I cut my work substantially short.
- I successfully created almost all of the core behaviour required for any imaginable direction of the game as it stands. Highways, blob sites, societies, construction zones, and tube upgrades are all critical pieces to any imaginable puzzle, and I have them all implemented and unit tested, with only a few missing tests.
- I didn't try to re-engineer the architecture I was using even once. When I encountered a problem that my current design didn't immediately address, I went for the simple option rather than something that was more complex. I even stayed true to the MonoBehaviour-heavy factory system despite the inconveniences it presented.
- I didn't focus on issues of performance, instead concerning myself with comprehensibility and ease of coding. This allowed me to create greenlit implementations with considerably less fiddling.
- The abstraction I placed between player commands, the implementation of those commands, and the state that results has left my architecture considerably less coupled than it otherwise would have. The UI is completely unconcerned with the structure of the game logic. The SimulationControl is largely unconcerned with the UI, and the game logic has only a single point of connection with the UI (pushing summaries to UIControl) and none at all with

SimulationControl. This means that I can modify the UI and the game logic drastically without having any meaningful effect on the other.

- I had a clarity of purpose for most of the development process. I could easily trace every piece of code I was writing back to the things the player needed that code to do for them.
- I managed to, for the most part, catch myself before wandering off into meaningless distractions, coding or studying things that had no relevance to the task at hand. There were very few tasks I performed, if any, that weren't immediately important to either the implementation of game logic or the creation of tests for that game logic.
- I figured out a sensible interaction between the various components of the game with little effort, one that has already proven reasonably effective at addressing the problems the game presents.
- Using TDD to consider the parameters of a program before beginning to implement it allowed me to detect problems and unforeseen consequences earlier in the process, which allowed me to fix those issues without having to fix or break down existing code.
- All of my unit tests give me a de-facto set of documentations that explain how everything interrelates. That way, if I'm ever curious as to how a given module functions, I don't have to trawl back through the implementation code to figure it out. I can just check the unit tests.

What could be improved?

- Several suites of unit tests are heavily coupled to specific implementations, which makes them brittle in the face of external change. Ideally, every unit test should use mocks of every class that isn't being tested. Even if that takes longer to implement, it'll make regression testing and test maintenance a lot easier.
- The way in which I prioritized various tasks led to an incomplete product at the end of the sprint. The fact that I focused first on all of the fundamental components, then on the integrations of those components, and lastly on using those components to make the game meant that no partial solution resulted in a deployable game. I think I need to split the sprint into atoms of player activity in the same way that the project itself is split into sprints.
    - For instance, BlobHighways and BlobSites represent an atom, of sorts, of player interaction. BlobSites with blobs and Highways with working priority and permission settings represent a unit of player interactivity: the ability to control where blobs move to. If I had focused on creating that entire stack (BlobSites, BlobHighways, the MapGraph, BlobDistributor, and all of the requisite UI) before implementing other things (Societies, ConstructionZones, HighwayUpgraders, ResourceDepots, etc) then I could at least have deployed some portion of the sprint's design goals, rather than the none I deployed this time.
- I need to determine exactly what amount of work each unit test is checking for. For instance, if a method has multiple effects and side-effects, do I create a single test that checks for all of them, or multiple tests that each check for one? I know right now it's fairly inconsistent, with some collections of tests defining very atomicized tests while others have a smaller number of larger tests.
    - I would imagine that the more subdivided paradigm, while taking longer to program, is more easily made complete, since it defines exactly the behaviour it needs and no more. In order to do that, however, I'm going to need a better naming schema, so similar tests are grouped together in space within the test results tab.
- While the practice I've gotten from doing full TDD for this prototype are useful, they were counterproductive to the ultimate goal of the sprint: making a prototype to see how the results of the paper prototype hold up in real time. Though I did begin to focus a bit more on creating

code fast rather than making it run perfectly, I need to keep pushing that, keep pushing to fail faster and get the barest minimum of functionality I need to test the experiment at hand. That might involve skipping TDD while I'm building prototypes and going for a quick-and-dirty solution. In fact, that's probably exactly what I need to do. TDD is for production code. Hack-and-slash is for prototypes. Even though practicing TDD here was useful, I need to do more hack-and-slash.

- I need to think more about the precise scale of work I can manage, and what work counts as supplemental. I ended the sprint with an estimated 8 hours of tasks I considered critical but unfinished, 9 that were desirable but unnecessary. I need to more formally divide tasks into Critical and Desired categories, and need to get a better grasp on what I can manage to complete in a two-week period. Most of that will be up to understanding the problem at hand and gaining experience with programming in general.
    - I might also consider trying to explain why I think a given task will take as long as it will. That way, I can compare my reasoning with the realities of the situation and get a better idea of what I was wrong about. That might increase the amount I learn per cycle.
- I need to determine how to use risk mitigation. After it was completed I rarely went back to it. I don't know if that's because the list itself had very little utility or if I simply failed to heed the advice it gives. It seems to me that the tasks in the sprint backlog should flow directly from the problems addressed by risk mitigation, that I should know exactly what risk I'm trying to mitigate or what problem I'm trying to address with each task. That paradigm leaves little room for unexpected problems, though, and it also might be unnecessary. Perhaps risk mitigation is just a planning measure use to set forth the initial task list, not something to be frequently reviewed and reconsidered.
- I need to try and extract more working hours out of my day. More work per week means that sprint deadlines are easier to make, and also means that I make more progress more quickly (assuming that I don't overwork myself). I can certainly shave off a few minutes here and there: shorter breaks, an earlier start time, more willingness to push for pomodoros 13 through 16. I could also try and positively affect my diet, which should help me stay focused. And of course building up my discipline will allow me to push aside lethargy and laziness more easily.
- I should try underestimating the amount of work I'm capable of doing so that I'm guaranteed to end a sprint with a deployable product. Not having something concrete to show for one's work is frustrating and demoralizing, and I would benefit much more from completing too small a chunk of work too early than too large a chunk of work too late. Plus, trying to reduce the amount of work I need to do will force me to figure out what really matters, which is a good thing for the design in general.
- Given that I found TDD so useful for specifying and clarifying the parameters of work, I might consider a hybrid of TDD and more traditional programming. I could continue to define the set of tests to properly analyze an implementation, but implement only some of them. That way I know the parameters of a more complete version of the work but can focus only on the tests that code behaviour I need for the immediate experiment. The others can throw a Test Ignored signal or some such. That way, I can define the high-level goals of a complete module (which allows me to do sanity checks on its functionality) but only implement the things I need (saving me time). It might be something worth considering.