Sprint 1 -- 2 Week Duration (Ends 3rd of March, 2017) 48-72 hours of work.

Basic Design

Story
You are some sort of abstract entity given jurisdiction over the transportation network of a large, underdeveloped rural region. It consists of nothing more than wilderness and farmland interspersed with a few villages. Your job is to develop this region in any way you desire, ultimately building it up into a vast economy of cities and metropolises producing a wide variety of goods and providing you with rather a lot of money.

Mechanics
The game world is organized on a large undirected graph. Some graphs have societies on them, while others are merely empty land. Players can build certain structures, like depots, on empty land, and can build tubes/railroads/highways/whatever connecting societies and structures to each other, or societies to other societies, or structures to other structures. Every society has some level of development (farmland, village, town, city, metropolis), ordered by complexity, which determines what sorts of resources it produces and what sorts of resources it requires. As demands are satisfied, societies will slowly advance up the complexity ladder, and as their needs go unfulfilled they'll slowly descend down the complexity ladder. It's the player's job to figure out how to advance as many societies as high as they can.

Aesthetics
I don't have a good way of answering or addressing this. I'd prefer a clean, pretty art style, but most likely the aesthetics will remain polygonal shapes for the foreseeable future.

Technology
Unity, deployed to PCs.

Risk List

1. I don't know how many levels of complexity there should be, how many types of societies there ought to be.
2. I don't know how many resources the game should have, what numbers make the game run or make it overwhelming.
3. I'm not sure what types of resources all the various societies should produce and what they should consume. I'd assume that farming societies produce food, but what do they consume? What differentiates a village from a town from a city?
4. I don't know how fast societies should expand and contract.
5. I don't know where players will manage to make meaningful choices. Are there tradeoffs between having a large number of less-complex societies and a smaller number of more complex societies? How should the game change between these two strategies?
6. I don't know where the challenge in this game is going to come. Should tubes cost money to create, and should money be somewhat difficult to generate?
7. The undirected graph might not facilitate the structure of the game, instead restricting meaningful activity.
8. I don't know what the victory conditions are, or whether there should be any. What's the player's motivation for playing?

9. The story might not be substantial enough for people to latch onto. It might be that I need a more concrete, more comprehensible conceit to encourage player engagement with the system.
10. I don't know if every society of a given complexity should produce the same resources. Will that lead to less interesting gameplay?

Risk Mitigation

1. I don't know how many levels of complexity there should be, how many types of societies there ought to be.
    a) It'll probably take some time to figure out how many levels of complexity to make, though I suspect it'll need to be more than two given how important the interplay between various levels of society seems at this point. I'll start with three--farmland, village, and town--and see if we can get any interesting interactions out of that. I'll see about building a paper prototype to test resource interplay between these three and work from there.
2. I don't know how many resources the game should have, what numbers make the game run or make it overwhelming.
    a) It's probably best to start with the simplest possible configuration and work up from there. We'll start with a single resource from every society type and see if that allows for sufficient complexity that we can see a kernel of fun. We'll save additional resource types for future sprints.
3. I'm not sure what types of resources all the various societies should produce and what they should consume.
    a) We'll start by testing a simple arrangement that should allow for reasonable complexity with few moving parts. Farmlands will produce food (as one would expect). Villages will require food to advance and will produce a second resource, and cities will require both food and the village resource and produce a third resource. Farmlands won't degrade (since they're the lowest-complexity society) but will require both the village and the city resource in order to advance. That way, villages rely on farms, cities rely on farms and villages, and farmlands rely on neither but are upgraded only when both are present. I anticipate this will create a tradeoff between the number of farms you wish to advance up the complexity ladder and the number you want feeding your cities. My hypothesis is that this might create hard lose states of all of the map's villages decay, at which point advancement becomes impossible. We'll need to work on that.
4. I don't know how fast societies should expand and contract.
    a) That is a fine-tuning question that we won't be able to answer for quite some time. We'll fiddle with turn timing in the physical prototype. If we get to a digital prototype, we'll have to fiddle with the numbers, but now we'll err on the side of caution and make expansion and contraction times somewhat longer than perhaps they need to be, so we have enough time to reason about the problem. Otherwise, this is an answer for external playtesting to solve.
5. I don't know where players will manage to make meaningful choices.
    a) With the model hypothesized in mitigation 3, we should have a reasonable choice at the level of farm advancement. There will probably also be intelligent choices at the level of depot placement in order to maximize efficiency. The paper prototype should provide some answers as to whether these choices are meaningful or not.
6. I don't know where the challenge in this game is going to come.
    a) Again, with the paper prototype, the obvious sources of challenge are in depot placement and advancement priority. However, it remains to be seen if these challenges matter without tubes costing something. There might be interplay between travel times and expansion/contraction rates, but we'll probably need a programmatic prototype to reason

about that effectively. We'll leave tube costs for a future sprint.

7. The undirected graph might not facilitate the structure of the game, instead restricting meaningful activity.
    a) Another difficult question to answer in isolation. We'll probably leave this one unaddressed for now, as we have no specific evidence one way or another that a rigid graph structure is going to hurt us. It certainly makes reasoning about the problem simpler. If we get stuck at some point, we can reconsider the graph assumption, but for now it'll have to be an acceptable risk.
8. I don't know what the victory conditions are, or whether there should be any. What's the player's motivation for playing?
    a) We're a long way from victory conditions. Let's first try to establish this system as a toy, without rigid motivational structure underlying it. We can figure out victory conditions once we've uncovered the core of our design.
9. The story might not be substantial enough for people to latch onto. It might be that I need a more concrete, more comprehensible conceit to encourage player engagement with the system.
    a) Another acceptable risk for now, given that we have limited ability to test or improve upon the aesthetic properties of the game. To be addressed in a future sprint.
10. I don't know if every society of a given complexity should produce the same resources. Will that lead to less interesting gameplay?
    a) This is a very important question, but for simplicity's sake I think it needs to be left on the back-burner. Once we have a reasonable core we'll address this problem again with another prototype.
11. What is the precise way in which societies ascend and descend the complexity ladder?
    a) Begin with a simple paradigm: a society has a certain amount of needs per time increment. If those needs are satisfied in full, increment some counter by one. If they are satisfied by less than half, decrement the same counter by one. Otherwise, the counter stays the same. If the counter reaches zero, the society descends. If it reaches some predefined number, the society ascends. Try and make the halfway points integer values for ease of prototyping.
    b) Or, simply ignore the timer altogether and experiment with that in the digital prototype.

Task Backlog

Completed Tasks

1. Set up the paper prototype with the rules specified in risk mitigation. [2 hours, completion time not recorded]
2. Brainstorm about what the behavioral requirements are for the core society type interactions. [2 hours, completion time not recorded]
3. Use the paper prototype to iterate on the basic loops of the game [13 hours, completion time not recorded]
4. Clean out all of the unnecessary modules in the codebase. [1 hour estimated, actual time 25 minutes]
5. Build the interfaces necessary for paper prototype 2.2, or change existing interfaces to meet its needs [3 hours estimated, completed in ~2 hours, possibly incomplete].
6. Devise a way of decoupling game logic from Unity's engine, to facilitate decoupled unit testing [estimated 2 hours, completed in ~1 1/2 hours].
7. Build a set of unit tests that prove the functionality of a Tube implementation [estimated 3 hours, completed in ~4 hours].
8. Create an implementation that satisfies the Tube unit tests [estimated 3 hours, took ~45

minutes].
9. Build a set of unit tests that prove the functionality of a Society implementation [3 hours estimated, actual time ~5 hours].
10. Create an implementation that satisfies the Society unit tests [3 hours estimated, actual time ~4 hours].
11. Develop a sensible software architecture that handles the exchanges between UI, game logic, and the underlying rendering/serializing engine [3 hours estimated, actual time ~ 8 1/2 hours; about 3 1/2 hours of research, 5 of drawing diagrams].

Tasks left to do

1. Lay out the full skeleton of the current program, with all of the interfaces and all of the incomplete implementations placed into files [2 hours].
2. Build unit tests for ResourceDepot [2 hours].
3. Greenlight the unit tests for ResourceDepot [1 hour].
4. Integrate the existing systems into the new architecture [3 hours].
5. Develop the interfaces and integration tests for the exchange of resources between societies mediated by highways (5 hours).
6. Use the new tubes and new society modules to turn paper prototype 2.2 into digital prototype 2.2 [5 hours].
7. Use the digital prototype to experiment with production and ascent/descent timing. [5 hours]
8. [To Do] Build another paper prototype to experiment with alternate productions within the same level of complexity and see what effect that has on the base game [13 hours]

Review

There is no digital prototype to preview. The sprint was killed on the 3rd of March, at 9:50 AM, due to structural pathologies in the programming methodology that led to no meaningful progress.

Tasks left incomplete (discounting ones that were removed prematurely):

1. Lay out the full skeleton of the current program, with all of the interfaces and all of the incomplete implementations placed into files [2 hours].
2. Build unit tests for ResourceDepot [2 hours].
3. Greenlight the unit tests for ResourceDepot [1 hour].
4. Integrate the existing systems into the new architecture [3 hours].
5. Develop the interfaces and integration tests for the exchange of resources between societies mediated by highways (5 hours).
6. Use the new tubes and new society modules to turn paper prototype 2.2 into digital prototype 2.2 [5 hours].
7. Use the digital prototype to experiment with production and ascent/descent timing. [5 hours]
8. Build another paper prototype to experiment with alternate productions within the same level of complexity and see what effect that has on the base game [13 hours]

Retrospective

The first two-week sprint experienced what I would consider a catastrophic failure in its latter half. Though the early sections went fairly well (the paper prototyping quickly narrowed in on some

interesting interactions) and early TDD worked fairly competently (there were several useful suites of unit tests that became implemented) this all started to break down in the second half of the sprint. Concerns over sensible architectures led to considerable study on the proper architecture of software, which led into a sudden Big Design Up Front issue right in the middle of the sprint. This BDUF proved wholly inadequate for the problems at hand, and was constantly being "fixed" as I started even the most basic implementation tasks, as the planning proved to be entirely useless. And, of course, it distracted me from TDD entirely.

I think what happened most strongly was that I got distracted by that siren song of overarchitecting that absolutely murdered the ever-living fuck out of Merchant Republics. I became so concerned with programming things the right way that I started burning more time figuring out how to program well than actually programming anything. Hours of my time fell into planning superstructure and foundations that were ineffectual and unhelpful. Realizing that the architecture was so insufficient caused my morale to plummet, which caused me not only to freeze up while attempting to program things, but to actively turn away from programming altogether. The 2nd of March had no programming work in it at all, since I was so dispirited by my literal backwards progress that I wanted nothing to do with programming on that day. The 3rd of March, too, was consumed fairly early by that sinking feeling that I was accomplishing nothing and for no purpose.

I think I'm having a problem internalizing the core concepts behind two things.

One is incremental, modular development. I desperately need to figure out how to define specific, enclosed programming tasks that can be constructed in a reasonable amount of time. I can't build the entire prototype all at once, or else my mind will simply start to melt. I need to much more carefully decide what the pieces are, and then develop those pieces in isolation as much as possible. I also need to stop concerning myself so deeply with the code being general or "right." Basically none of my experiments with architecture have been of any use. Architecture wasted months of my life on Merchant Republics, and fucking destroyed this sprint. Designing these things up front is useless. It's exactly what Agile techniques are trying to prevent. It's this exact tendency of mine. I need to stop this or I will never be able to program anything effectively.

The second major problem is related, but I think worth addressing independently. That is my understanding of prototypes. I lock myself into these insane notions of what a prototype ought to be, coming up with these over-elaborate coding structures that might be appropriate for production code but are woefully inappropriate for a FUCKING EXPERIMENTAL PROTOTYPE THAT'S DESIGNED TO BE THROWN AWAY. Up to this point I've been hiding behind pedagogical arguments, claiming that I'm overdeveloping the prototype in order to "learn how to code properly." But I've realized (and hopefully I've internalized) that that argument is a fucking fabrication. Coding properly is coding at the speed and level of rigor that's required for the task at hand. Rigorously engineering a prototype reinforces bad habits. It is bad programming. It is learning bad programming. I need to realize this. A prototype is a prototype. I've got to stop wasting my time like this.

But identifying the surface-level problems is only part of the issue. The real question I need to answer is why. Why do I do this? Why do I compulsively over-engineer? It essentially amounts to overplanning, I would guess. Set out everything ahead of time to make future work easier. But I don't think I've ever been a compulsive planner. Looking ahead into the future in a rigorous and plan-oriented way has rarely been a pathology of mine. So what is it that's causing my mind to fall into this trap?

It might be ego. Maybe I wish to view myself as the Ace Programmer, and using fancy structures and techniques is one way of trying to prove that. Maybe I simply don't want to program, and building these architectures is a way of postponing that task as long as possible. I've certainly spent a lot of time actively idling while reading about things that probably won't help me in the long-run, but that are ostensibly related to the topic of programming.

There's also my concern with lack of structure. My life has never been particularly structured or rigorous. It's only recently that I've been enforcing some sort of schedule. Building these elaborate architectures could be a fulfilling an obsessive need for structure. Certainly my lack of social contact and context leaves me without meaningful structure that most other people would have. And I did enjoy creating that loose, useless, and inaccurate pseudo-UML diagram. It was satisfying while I was making it, to be sure.

Or it could be that nothing much special is going on. I have simultaneously taken on Scrum, the Pomodoro Technique, an Action Plan, a Work Diary, TDD, and a complete shift of goals on Strategy Blobs in the course of, what...17 days? I started all of this on the 14th of February (which I now know because I've been recording almost all of my work). It has been a paradigm shift in the way I organize my work and my life. Of course there were going to be stumbles. Of course the first sprint was going to be rocky. And it's not like it was a complete failure. We did some TDD, and the paper prototype was very successful. Not to mention that failure isn't a negative thing. It's how one learns. Maybe if I keep saying that enough, my heart will believe it in the same way my mind does.

So...how can we improve? I think there are several ways.

1. **Stop BDUFing:** No architecture that I plan out will ever survive contact with the realities of the problem. That is true for veteran programmers, but it is especially true for me, who knows so little about how this is all supposed to work. I need to completely drop the idea of developing my architecture as anything more than an incremental process. Build it as I need it.
2. **Break up the Problem More:** A "Skeleton Walk" seemed like such a simple thing to do. 2 hours. Until I realize that Skeleton Walk isn't an atomic task. Not by a long stretch. I really, really need to focus on breaking down everything I'm doing into smaller chunks, and then breaking those down into smaller chunks, until ideally I have Pomodoro-sized bites of work that I can eat away at. Methinks that most work cycles should end in blue, orange, or red text.
3. **Concern Myself Only With Deployable Mechanics and UX:** Units of work should be composed of mechanical sets. Building Highways is some discrete task. Building Societies is some discrete task. Building BlobSites is some discrete task. Building the integration that connects highways to societies is (probably) a discrete task. Building the UI that allows people to draw highways between things that can accept them is yet another discrete task. I need to focus on the things that the end user needs, and create understructure only when it absolutely, positively must be constructed to satisfy some use case.
4. **Make Some Large-Scale Decisions Up Front, Then Stick with Them:** Doing architecting in the middle of the sprint was suicide. There are definitely some things that need to be thought about, but they've got to be front-loaded as much as possible. I need to reduce the total number of things that I'm thinking about at any given time, and the things that I'm most susceptible to are architectural changes, so I need to force myself to stick with implementation plans the whole sprint through.
5. **Make changes through formal refactoring, but only after the feature is complete:** Refactoring is important, and it must be the main way in which I change existing structure. But refactoring can only be done once the modules in question greenlight all of their unit tests. I

should endeavor to stick with planned design decisions, no matter how messy they are, until I've implemented the mechanic, after which I can go back and create a formal task to refactor them.

6. **Individual mechanical tasks must be developed in isolation:** For this sprint, one of the first things I should've done is created a bunch of branches in my repository: Society, Tube/Highway, ConstructionSite, etc. These units should then have been defined in interfaces, given testing suites, and then implemented in as much isolation as possible. Only once those units have green-lit all of their tests can they be meaningfully reworked into the main branch. Hell, they should probably have their own namespaces and directory paths. Assets.Society, Assets.Tubes, Assets.Construction, maybe? I'm not entirely sure how that'll handle more integrated things (like UI), but it's certainly a start.
7. **Try not to get frustrated by failure:** This shit's hard. There will be a lot of failures. The failures are lessons; an indication of future strength rather than present weakness. I must insist to myself at every instance of disheartening frustration that this is all part of the path to excellence, that I am made better by my failures. The fact that I'm fucking up means that I'm exactly where I should be.
8. **Get better at throwaway code:** Some code is designed to last. Other code is designed to be thrown away. The next time I try to implement Paper Prototype 2.2, it must be throwaway code. Quick and dirty. That way I can test the thing that was supposed to be tested.
9. **Meditate away negative emotions:** There is a reason I've been so dedicated to the idea of meditation. It is a way of gaining control over my own mind, over my own self. I should use it, to the best of my abilities, whenever there is frustration that makes it difficult to think and act. I can eliminate this nonsense. It will take a long time, and constant vigilance, but it is possible. More than that, it is inevitable if only I keep going.
10. **Do not lose site of the goal:** You do this for a reason, a vision in your mind of what you want to do and who you want to be. Don't forget that. Do not let the tangents, the distractions, the bias and bullshit claim your future. It is yours to build. not theirs.