CSCI 5920

11/29/2021

Matthew Michaelis

Final Report

GitHub: https://github.com/matt-mich/ember

(Note: A new demo video can be found here that shows off some of the new features: https://youtu.be/Z74-fmcxhAo)

(Sorry that the 'addressable questions' are just in a list. The requirements for the paper were not available for many days just prior to the deadline, so I wrote primarily in a different format, and had to stitch them together like Frankenstein's Monster)

What you are most proud of about your game?

My favorite part about the project is the effective synchronicity between the two versions when run simultaneously. If you do a task on the HTML version, the same resources show up on the UE4 version at nearly the same time. Making that happen has been a massive undertaking, much more so than I originally anticipated, but I've learned so much about data management and network synchronicity that it's more than worth the time I've put in. I also really like the fact that now that all the core elements are in place, I can start working on the actual fun parts of the game, stabilize it, and make it playable as more than a demonstration, but actually create a fun experience out of it.

What changes you made to your original game design for technical reasons and why.
The main change that had to be made was the scope had to be reduced. I was over ambitious, and because of that I had to make some sacrifices, namely in progression and storytelling. Since the net code took dozens of hours on its own, I didn't have the time to utilize the framework in a satisfying way as to match the original scope.

What changes you made to your original game design for playability reasons and why.
The game stuck close to the original intention in terms of playability. There's a First-Person modality (UE4), and a 2D Third-Person modality (HTML), which was pretty much all that was conceived of at the time of developing the original concept.

What did you learn from your play-testers?
Since I don't have very many friends, my Fiancé was the only person who played the game aside from myself, and while she didn't hit any major roadblocks, her major issue was that there wasn't much of a game to speak of after the novelty of seeing what you do on one screen affect the other wears off. It's a game framework, and while there are things you can do, it's more of a demonstration of the technology of asymmetrical

What changes did you make to your game as a result of the play-testing?
The play testing revealed some bugs in the net code which had to be resolved. I assume there are many more bugs lurking in the shadows because of the complexity of the program (The server alone is over 600 lines of code, while the HTML game is over 1300), and contains an SQLite database and a JSON world archiving system.

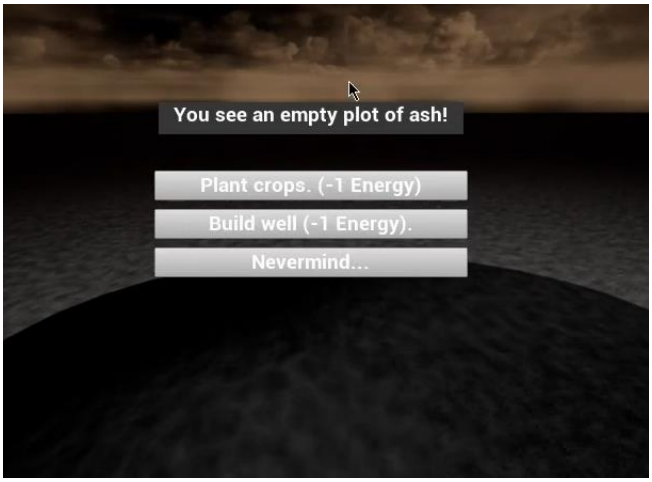
What you would do next if you had more time?
I would just keep developing what is currently a tech demo into a full game. I'd put timers on the tasks so you couldn't just spam them for infinite resources. I fully intend to do so and develop this game into something really special.

What you would do differently next time?

Part of me would want to rework the net code to be less reliant on constant server queries, and instead establish a continuous connection to speed up the process. Currently, much of the system in UE4 is slow (You can see this in the fact that when you walk up to an interactable object, the options prompts take seconds to appear, and sometimes never do.) because of the reliance on a REST API system rather than a genuine net code solution.

How to play:

The game is relatively simple to play. Once you've logged in (See instructions on the GitHub page), you can wander around the Ashlands, either by using the arrows to scroll around in 2D to look at each of the cardinal directions, or by walking around in 3D using the standard WASD/Mouse-look scheme in UE4. You interact with objects by clicking on them (HTML) or by walking up to them (Sometimes you have to back up and then walk up to them again as the server trigger can be a bit finicky for some unknown reason no matter how much I bash my head into the brick wall of net code). Once the dialog has loaded, you can select any of the options given to you, which will cause an immediate action to take place across both game worlds. In the screenshots below, you can see the difference in what it looks like to interact with a blank plot of ash in both versions of the game.

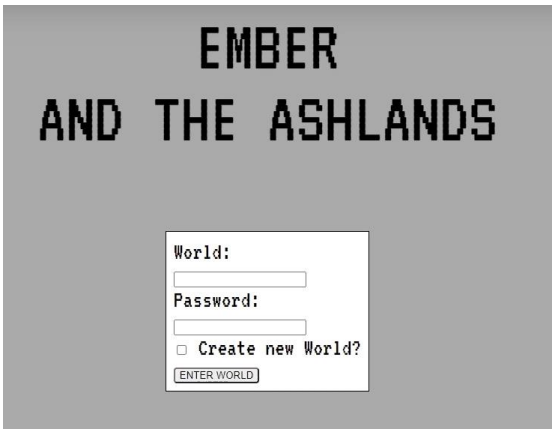


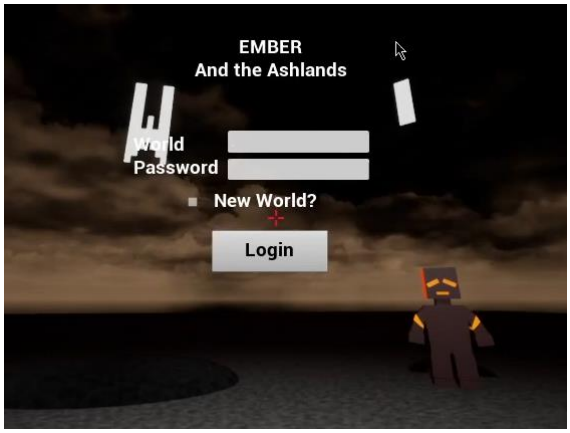
The same goes for characters, of which the game unfortunatly only has one when I intended to have about five. He doesn't say much, but you can feed him to boost the morale resource (Which is useless at the time of writing. You should still feed him, though.

Additionally, pressing 'L' in the UE4 version will allow you to login to a different world, or even create a new one entirely by checking the relevant square. (See screenshots in next section)

How it works:

The game requires a login, which it uses to secure each environment. These environments are saved as files which are loaded whenever a successful login occurs. Logins and (hashed) passwords are stored in an SQLite Database.





Each world/environment file contains information regarding the player's game status. This includes what constructions they have and includes a log of what tasks the player has completed, and when they were done. This is useful for when Ember needs something to say (Or at least this was the original intention of it, Ember's dialog doesn't change in the current iteration which would be resolved if I had more time), at which point they can either congratulate the player for doing something (or for a consistent pattern of doing the thing indicating a habit being formed) or they can recommend something that hasn't been done in a while. Information regarding Ember can also be stored such as how much he has been interacted with, how much food he's eaten, and any story information. The last piece of data currently stored is the player's resource values. All information is automatically synchronized between the various platforms and updated in real time.

The synchronization is done through usage of the Python based Flask server which hosts the game's API. The API itself has two primary modalities; ask and tell. Ask requests lead to the server responding with relevant world data, while tell requests are used to update the server. There's also the tell-ask, which is used for when the client doesn't know how an action will affect the world data, such as if a task is completed, it will tell the server that it was done, and waits for confirmation that resources have been awarded.

This might not be the most efficient solution, and if I've learned anything over the course of this project, it's that net code is hard. Net code probably shouldn't be implemented with API calls. Proper net code would require a constant data stream, as I mentioned before, but I thought this would be simpler. The key letter in that word is the 'r', as this turned out to be anything but simple.

Questions kept arising regarding efficiency at scale. Things like "Should the server, for each active session, store the JSON world file in memory? Or should it access the file every time it needs to check for changes compared to the most recent update from a game instance?", which seemed easy to answer, like the fact that multiple sessions could modify the same data, so it should be a file and not stored in the session. Then the thought would arise that multiple sessions could access the same data segment in the server's memory if they share a session key for the login. This would make it so you could eliminate extraneous file reads but increase the memory footprint of the server as it would have to effectively hold all the world files until they're needed,

which could be fixed by only holding recently accessed worlds, and purging non-active ones, after writing their last status into their respective files, of course. This led me to the conclusion that simple questions become very large at scale, thus they do not scale efficiently. I could have easily made choices that would allow for less work, as this project is unlikely to be scaled to hundreds of game instances, but I wanted the practice of net code management, so these thoughts were constantly indulged with overthinking followed by semi-coherent, potentially scalable implementation.

Additionally, security concerns arose during the implementation of various systems. Most of these revolved around trust of the client. Since it's JavaScript, the user could modify the game such that it grants them, for instance, infinite resources, and simply report back to the server that it has them, even if there was no mechanism for them to gain so much in such a short period of time. This led to an increase in complexity for these systems as everything that needed to be trusted should be done on the server. In the ideal case, the user client would only report what actions the user wanted to take and would validate those actions server side before distributing resources. This would mean that in order to complete a time-delay task such as doing the dishes (Which should only be done around once a day, if that), they would have to report to the server that the task has been done, at which point the server would check to see the most recent instance of the 'doing the dishes' event was around a day ago, at which point it would award resources. If not, it would have to send some error (Though the unmodified client shouldn't let the button be pressed in the first place since it also stores a log of events, thus knowing when each event occurred, and logically when the next is allowed to occur). At some level, every world modifying action should go along with a server request, like the client asking its parent if it's allowed to go to the movies, but not something that doesn't matter, like walk around the house.

In total, the main thing that I learned was that game design is incredibly complicated when combined with network data management. I've designed games before, but none of them had this level of interconnectivity which required a tremendous amount of research and debugging before it became playable. And after everything, the game itself if not fun at all. I suppose there are plenty of things I can do to make it fun, or engaging, but what I feel like I've created is a foundation of a solid game rather than a vertical slice of a complete experience. More than anything, it motivates me to build more upon it; add minigames, add the story elements that I planned from the beginning, and add more progress that a person can make to keep them engaged and coming back to the application.

How to run it:

Please see the GitHub README for detailed instructions on how to install/run the game. Note: You will need Python 3.7, Flask for Python, Flask_Session for Python, and VaRest installed into UE4 through the Unreal Marketplace.

References:

(Sorry for the lack of references. I had to write this blind to the requirements due to the system outage, so I didn't collect them.)