**Arithmagician Architecture and Code Spike**

**High Level System Architecture Description**

Arithmagician uses a 256x256 world file to represent the initial state of the game world. The entire world is contiguous and the are no separate rooms or areas, allowing for global interactivity of game elements. This large world file will be stored centrally for all players in an AWS S3 bucket as a binary file. Once downloaded, JavaScript in the browser will restructure the world file into arrays of UIint8Arrays, a string array, and world state flags and variables. The same initial world file will be served to all players.

A player’s three save files are much smaller than the 256x256 world file and will be stored for each player in an AWS DynamoDB database. These files will contain flags representing the persistent changes to the world state, and an array of point-to-point transformations of the object positions the player has effected. Save files will be protected with AWS IAM and Cognito.

To minimize the cost of cloud storage, new players will download the world data (large), and then begin building their save files (small). When the player logs in he will first check his local cache for up-to-date game assets and save files. If the save files are not present or are out of sync with server files, he will download his save file from the cloud. The save file will provide instructions to rearrange the static game world into the appropriate state for the player.

The frontend portion of this project is split into three separate apps:

**Editor**

Allows the developer to create, load, and save world files

Uses simple HTML input and display controls

Uses an HTMLCanvas and a static renderer to display the current world file

**Tester**

Allows the developer to quickly test a world file from the editor

Is not associated with a player account

Does not persist data

**Game**

Allows a player to create an account or login

Loads or fetches the player’s save file

Periodically saves the world state, and saves on exit

**High Level Code Architecture**

The editor uses an HTML canvas and input/select control to manipulate data on layered 256x256 data grids. These grids are arrays of UInt8Arrays, allowing for ease of cell indexing and relatively compact storage. The editor allows the player to place objects, connect objects, modify object data, and copy/move/delete rectangular regions of data. Arrays of changes are pushed in batches to a modified stack to allow the developer to undo or redo changes.

The editor’s renderer does not implement shader effects. Because an editor can zoom/pan arbitrarily, the rendering of many elements (up to 256\*256\*3 layers + text overlays = 196,608+ image draws) must be fast. Fast image rendering is achieved by creating large static tilesets from small image templates, and performing fast draws to the canvas using indexed sub-images from the tilesets. Wherever reasonable, images are pre-rendered to reduce rendering logic and GPU load.

The tester and game apps both run the same game engine. The engine captures the player’s input, uses recursive traversal of objects on three layers to determine movements, updates the positions and states of game objects, and renders the current scene to a full-screen HTMLCanvas based on the player’s position. The game app additionally authenticates users and persists their progress to local storage and the cloud.

The tester and game renderer draws a fixed 16x9 cell viewport and does not allow for arbitrary zooming and panning. As such image drawing in the tester and game are not a performance bottleneck (at most 16\*9\*3 = 432 cell images need be drawn). The tester and game apps will instead use the player’s available graphics resources to run shader code using an WebGL shader class designed to run shaders at a fixed display resolution (960x540 pixels) given an HTML canvas and a fragment shader program string written in GLSL.

The current set of shader effects will be determined by the state of game variables and flags, and by the cells in the current viewport. For example, if a snowy tileset is prevalent in the current viewport, the shader manager will run a program to render a snowy overlay on the screen. Ambient sound, music, and volume will also be determined by the prevalence of different tile types and events on the screen.

**Database Schema and REST API Description**

The database and file storage schema are simple, while the contents of filetypes served will be more complex. Each player account will store a username, a hashed/salted password, and 3 save files. The node.js Elastic Beanstalk server will authenticate users and then serve them a save file based on their selection in the GUI. The static, undifferentiated game assets (.png, .mp3, and the static world file) will be served from an S3 bucket.

The browser will handle transforming retrieved data from compact storage formats (binary) to interaction-friendly in-memory data structures (grids), and performing the inverse transformations. The S3 bucket will serve static assets, the node.js server will manage authentication and database routing, and the DynamoDB will store save data using key-value pairs, with keys made of the user’s unique username and save file number, and save file data as the value. The upper bound on a save file’s size will be much smaller than the size of the static world file.