Dungeons of Dread, JavaScript Version

7/22 – I had thought about using P5.js to make handling the canvas graphics easier since I am very familiar with it, but then learned about a small library called rot.js that is specific for handling the graphics in roguelike games. So I downloaded the minified library into my project directory and created an index.html and a game.js file to test it out by drawing some text to the canvas using the ROT Display object.

7/23 – I created a Game object and gave it an init method to create a ROT.Display object with dimensions of 80 x 24 characters. I also created a getDisplay method to return this display.

I called Game.init in the window.onload method and appended the display using getDisplay and the ROT.getContainer methods to index.html.

The next step is to implement some screens for the game. Each screen will handle its own rendering and input processing, but I will need to create an interface to these screens in Game.js to handle entering and exiting the screens (switchScreen method) and dispatching events to the screens. This method takes a screen object and checks to see if there is a screen currently displaying; if so, the current screen’s exit method is called. The current screen is assigned to the screen object and its enter method is called. Next, the screen’s render method is called.

I created a bindEventToScreen function in the init method that adds an event listener and calls the current screen’s handleInput method if it detects a key press.

In screens.js, a Screen object is defined within the Game namespace. It can be of 4 types: a start screen, a play screen, a win screen, and a lose screen.

I implemented enter, exit, render and handleInput methods for the Game.Screen.startScreen object.

Game.Screen.startScreen.enter logs to the console that it started.

Game.Screen.startScreen.exit logs to the console that it exited.

Game.Screen.Start Screen.render uses the display object parameter and the ROT drawText method to render the game title and tell the user to press Enter to start.

Game.Screen.handleInput accepts parameters for inputType(keypress) and inputData(which key pressed) and switches the screens if applicable.

7/24: For now the enter and exit methods of each screen log to the console whether these actions happened. After the init function runs in Game.js, the Game.switchScreen.startScreen is called. The startScreen.render method renders the title (Dungeons of Dread) to the screen and instructs the user to press enter to play. startScreen.handleInput detects that enter is pressed and calls Game.switchScreen, passing the parameter Game.Screen.playScreen.

Each screen is implemented the same way with enter, exit, render, and handleInput methods. From the playscreen the win screen can be entered by pressing enter and the lose screen by pressing esc. Most of this code written so far today tests that the screens can be switched and displayed after the game is loaded.

glyph.js – handles glyphs which are the graphic entities in the game. They consist of a character, a background color, and a foreground color, which are parameters to the Glyph object constructor. I created getters: Game.Glyph.prototype.getChar(), Game.Glyph.prototype.getForeground(), and Game.Glyph. getBackground().

tile.js – Tiles contain a glyph (and eventually, other information) and a 2d array of tiles will constitute a map of the game space. The Tile constructor takes a glyph as a parameter.

I created an acceessor to return the glyph of a tile, Game.Tile.prototype.getGlyph. Three types of tiles were instantiated. Game.Tile.wallTile(#0, Game.Tile.floorTile(#) and a null tile that is returned if access out of bounds areas is attempted.

map.js – The map is constructed from a 2d array of these tiles. The Map constructor takes a 2d array as a parameter and also uses the row length as its height and the column length as its width. There are accessor methods: Game.Map.prototype.getWidth, Game.Map.prototype.getHeight, and Game.Map.protoype.getTile, which takes x and y coordinates and returns the tile at that position on the map. Tiles[row]{col] is mapped to tiles[x][y].

screens.js – the game map is created in the Game.Screen.playScreen.enter method. An 80 x 24 character map is created and its x and y coordinates filled with null tiles. Then the ROT.Map.Cellular method is called to generate an 80 x 24 array using the strategy of cellular automata, randomly generating 0’s and 1’s. When generating the map, 1’s are assigned to floorTiles and 0’s to wallTiles.

7/25

game.js – assigned screen width and height to variables instead of hard-coding them and used in the init method and wrote accessors for them.

screens.js – In Game.Screen.playScreen , I increased the width and height to 250 characters and used ROT.Map.Uniform to generate a more dungeon-like map, with 0’s set to floorTiles and 1’s set to wallTiles.

Game.Screen.playScreen.move – Takes x and y parameters and calculates \_centerX and \_centerY(the player position) making sure using Math.max and Math..min functions that the position move stays within the bounds of the map.

In the playScreen render method, I created 2 variables, \_centerX and \_centerY, both initialized to 0 in playScreen to use in centering the map to the player’s movement. The topLeft and topRight variables are calculated using these center variables and the screen dimensions to center the mqp on an @ glyph. handleInput will call this method, moving one character in the appropriate directions, based upon the arrow key pressed.

The render method is modified so that the map will be centered on the players movement and recalculate the topLeft x and y positions so that the screen will stop scrolling if half the screen can’t be fit on both sides of the player. Then the map is rendered starting from the calculated topLeft and topRight positions.

Finally a Game.Screen.playScreen.display.draw method draws the @ glyph at \_centerX – topLeftX and \_centerY – topLeftY.

The Glyph constructor was modified to be generic, setting name, foreground, and background.

The Game.Tile floor and wall tiles were updated to use this object and also given the property isWalkable.

entity.js – Besides the glyph, all game entities will have common properties of a name and x and y positions, so the entity will be the base object for all different game entities. In addition, entities will have additional properties/behaviors that will be implemented through the idea of mixins. For the Entity class I extended the Glyph class with name, x, and y properties and implemented getters and setters for these.

For mixins, a property called attachedMixins (object) is added. Mixins should have a name and an init method and shouldn’t override properties of the entity. The additional properties are copied over to the entity and, once the mixin is attached to the entity, its init method is called if it exists. Game.Entity.prototype.hasMixin is a method that checks if a mixin exists by passing in its name or the mixin itself.

map.js – Game.Map.prototype.getRandomFloorPosition returns a randomly generated floorTile’s x and y coordinates. This is where the player will start.

entities.js – will hold mixins for entity objects. The first to be created is Game.Mixins.Moveable

its name is Moveable and it consists of a method, tryMove that accepts an x and y coordinate and the map and tests if the tile located there is walkable, returning a boolean.

A player template object containing the player glyph and the Moveable mixin is created.

screens.js – is refactored to remove the references to centerX and centerY and replaced with a player object produced from the GamePlayerTemplate. In playScreen.enter the player’s coordinates are calculated from map.getRandomFloorPosition and the player is located there using player.setX and player.setY. In playScreen.render, references to centerX and centerY are replaced with player.getX and player.getY to center the map on the player. All tiles now extend glyphs so the display.draw method is also modified to use tiles.

A move function is also added to the playscreen that checks the change in position if the user presses an arrow key and adds this to the player’s current coordinates if the player.tryMove method returns true.

Now the player can move about the map which is continuously centered on his/her position.

7/26

entity.js - I updated the entity contructor to add a map field and created setMap and getMap methods so that other entities can be added to the map at specified coordinates.

I updated the entity constructor to have a groupName so that entities that have common behaviors, such as moveable, have an interface of common behaviors. An entity template for a python was created.

map.js – in the constructor, I created an array to hold the entities, and a scheduler using the ROT.Scheduler.Simple and ROT.Engine methods which order the entities into a queue so that they take turns in a certain order. The ROT.Engine extracts the entities form the scheduler queue and call the act function on any entities that have movement. After the engine is started and it is the player’s turn, it is locked until a key is pressed, and unlocked afterwords to execute the turn. The engine sequence is to:

1. Process entity turns until it’s the player’s turn.
2. Render the screen and lock the engine.
3. Wait for a key press, process the move and unlock the engine.
4. Repeat.

Methods getEngine, getEntities, and getEntityAt which takes x and y coordinates as parameters, where implemented.

game.js – The bindEventToScreen method is modified by adding a adding a refresh helper method that clears the screen and renders the current display. This method is also used in switchScreen.

entities.js – Mixins with a group name of Actor will be created for the entities which contain only an act method, allowing easy checking of whether an entity needs to be scheduled by the scheduler. The player’s act method causes the game to refresh and the engine to lock until there’s a key press when it’s the players turn.

The Moveable mixin is modified so that if an entity is at the position on the map passed to moveable, the tile is not walkable.

map.js – The methods, addEntity and addEntityAtRandom position add entities to the scheduler is they have the ‘Actor’ mixin and add the entity to a random floor tile on the map. The getRandomFloorPosition is also modified to check that the tile has no other entity occupying it. Then the player and several pythons were added to the map after creating the scheduler and engine.

screens.js – In the playscreen, the player is created from the PlayerTemplate and the map is created, passing in that player. Then the engine is started. In the render function, the entities (for now, pythons) are rendered to the map. The engine also needs to be unlocked in handleInput after the key is pressed.

7/27

map.js – A removeEntity method is created which splices the entity argument from the array and also from the scheduler if it is an Actor.

Since the python will be able to grow, a isEmptyFloor method is implemented which checks that the position for the x and y coordinate arguments exists and doesn’t have an entity (by calling getEntityAt. The getRandomFloorPosition is updated to use this method.

entities.js – A mixin object, Destructible is created with an init function that assigns it a number of health points(hp) and contains a takeDamage method which subtracts damage from the health points and removes the entity from the map if the health goes to 0 or less.

A mixin object, Attacker is implemented with a attack method which checks if its target has the Destructible mixin and calls takeDamage on the target, passing in the health points to be removed.

Both player and python templates are updated, both with Destructible and the player with Attacker.

The tryMove method of the Moveable mixin is also modified so that, if the entity that is moving(player) is an attacker and the move tried is occupied by another entity, it will initiate an attack.

The python will be allowed to grow up to 6 tiles, so a property, growthsRemaining is added to the PythonActor mixin. Every turn, whether it grows is determined randomly and if so, where it grows is determined by a random number between 0 and 1 for the x and y coordinates. These numbers are offsets added to its current position (1 tile in any direction) If that tile is unoccupied then a new python is created.