Super Mario Project : Bringing back the Arcades to your browsers

A PROJECT REPORT

Submitted by

CB.EN.U4CSE12144	**Srivenkata Krishnan**
CB.EN.U4CSE12450	**Travis Joseph**
CB.EN.U4CSE12244	**Sharad S**
CB.EN.U4CSE12155	**Vignesh R**

done as part of the CSE471 Free and Open Source Software course
for the academic year 2015-2016.

Contents

1 Introduction

This is the implementation of the popular **Nintendo's Super Mario** aracade game as our our project as part of the FOSS Course. This is a free HTML5 remake of Nintendo's original Super Mario Bros, expanded for the modern web. It includes the original 32 levels, a random map generator, a level editor, and over a dozen custom mods.

2 Development

This section discusses on the various tools, infrastructures and environments that was set up to build this web application.

2.1 Tools used

- SuperMario uses Grunt(http://gruntjs.com/) to automate building, which requires Node.js(http://node.js.org).
- Github is used as a common platform for bug-tracking, version control and also hosting the game online.
- Github pages(gh-pages) is used as the branch to create web pages that are hosted on Github The process is straightforward of development was straight-forward while using Grunt's help page(http://gruntjs.com/getting-started).

2.2 Coding Platform

SuperMario is built on a modular framework called GameStartr.The FullScreen-Shenanigans (https://github.com/FullScreenShenanigans/) organization contains GameStartr, its parent class EightBittr, and the modules used by the GameStartr framework. All these (theoretically) have their own README files, which you should skim before developing for SuperMario itself.
The FullScreenMario.ts class declaration contains class functions and some constants, while static settings to be added to the FullScreenMario prototype, such as map layouts and object attributes, are stored in files under the Settings sub-directory in the Source folder, such as audio.js and collisions.js.

3 Builds and Releases

SuperMario uses Grunt to automate building, which requires Node.js.
Go to the root repository folder and run the following commands:

```
npm install
grunt
```

You may also have to manually install grunt first:

```
npm install -g grunt grunt-cli
```

We have added full screen support and so all the files in the releases are named as **"FullScreenMario"**. Source/FullScreenMario.ts, along all the other files in Source, will be compiled into a new Distribution folder in the root.

- FullScreenMario.d.ts and FullScreenMario.ts are merged into a FullScreenMario.ts, and all .ts files under References are copied as well. A .js file with a .min.js and .min.js.map will be generated as well.
- Distribution/FullScreenMario-0.10.X is a minified version for use in static websites. It's also zipped for your convenience.

4 A Developer's Manual to this projeect

FullScreenMario (FullScreenMario.js) is the governing class. The global window.FSM is an instance of FullScreenMario, and everything in the game is a member of FSM. The FullScreenMario class itself inherits from GameStartr (Source/References/GameStartr-0.2.X.ts), which inherits from EightBittr (Source/References/EightBittr-0.2.X.ts).

The base GameStartr engine includes a large number of modules, all of which are stored in Source/References (AudioPlayr, ChangeLinr, etc.). The naming schema is to have two words, the second of which is a verb ending with 'r'. The class will have the ending two characters abbreviated to 'r', and the instances aren't abbreviated. FSM.ObjectMaker, for example, is an ObjectMakr instance.

Each module contains a class by the same name. The global ObjectMakr module, for example, contains the ObjectMakr class. To use it, you would call **var ObjectMaker = new ObjectMakr.ObjectMakr(...);**.

FullScreenMario is the same - **var FSM = new FullScreenMario.FullScreenMario(...);**. See Source/index.js for sample usage. It looks a little silly from the JavaScript side, but makes organizing the TypeScript interfaces and classes much easier.

4.1 Game World Design and Implementation

Everything you see in the game (trees, blocks, the player, etc.) is a Thing. The Thing class is subclassed by a new class for everything (Tree class, Block class, Player class, etc.). When added to the game, a Thing has a number of properties filled out. These include velocities (xvel and yvel), positioning (top, right, bottom, left), and so on.

Coordinates are relative to the top-left part of the screen. If you have experience with CSS, this is the same as positioning HTML elements absolutely. To add a new Thing to the game, use FSM.addThing("type", left, top):

FSM.addThing("Block") //Creates a new Cloud and adds it at x=0, y=0
FSM.addThing("Brick", 32, 64) //Creates a new Brick and adds it at x=32, y=64

4.1.1 ObjectMakr

All of FullScreenMario's non-GameStartr classes, including Thing and its subclasses, are defined in Source/settings/objects.js. In short, the class hierarchy is stored under FullScreenMario.FullScreenMario.prototype.settings.objects.inheritance

and the attributes for each class are stored under FullScreenMario.FullScreenMario.prototype.settings.objects.pr
You may read ObjectMakr's readme for a full explanation.

FSM.ObjectMakr.make("type") is how you make a new Thing in the game. It takes in a string for the class name, and optionally an object containing additional properties for it. For example: **FSM.ObjectMakr.make("Block");**
// Creates a new Block
// Creates a new Brick with a Mushroom inside
FSM.ObjectMakr.make("Brick", "contents": "Mushroom");

4.1.2 GroupHoldr

Each Thing has a groupType string property that determines what group it's considered to be in. These are, in order from visible top to bottom:

- Text
- Character
- Solid
- Scenery

FSM.GroupHoldr contains an Array for each of the groups; each Array contains all the Things of that type currently in the game. Things are added to their respective group when added to the game, and removed when they die. The groups are accessible both by static name and via passing in a String:
// Returns the Solid group **FSM.GroupHolder.getGroup("Solid");**
// Returns the first Solid, commonly a Floor **FSM.GroupHolder.getGroup("Solid")[0];**
FSM.GroupHolder.getSolid(0);

4.1.3 Triggers

The objects and map systems provide hooks for Things to have certain member functions called on them. Currently, these are:

- **onMake** - When the Thing is created (generally FullScreenMario.prototype.thingProcess)
- **onThingAdded** - When the Thing is first added to the game state

4.1.4 Movements

In order to progress game state and repaint the screen, the game calls FullScreenMario.FullScreenMario.prototype.upkeep() every 16 milliseconds (while running at 60fps). This is governed by FSM.GamesRunnr.

Inside upkeep, a maintenance function is called for characters and solids. These are FullScreenMario.FullScreenMario.prototype.maintainCharacters, and FullScreenMario.FullScreenMario.prototype.maintainSolids. During these maintenance calls, for each character and
solid, if they have a .movement property, it's called as a Function on the Thing. These will typically be FullScreenMario.FullScreenMario.prototype.moveSimple (such as Goombas) or
FullScreenMario.FullScreenMario.prototype.moveSmart (such as smart Koopas).

4.1.5 Sprites

Sprites are a sequences of graphics that when rendered together gives the effect of say,a ball bouncing off a surface, bird flying across the screen,etc.

Thing sprites are stored in Source/settings/sprites.js using a custom image format. The image format is discussed in more detail in PixelRendr's readme.

The key used to look up a Thing's sprite is computed using the current area name and the Thing's .title, .groupType, and .className (GameStartr.generateObjectKey). Whenever a Thing is added or has its class changed, its sprite is recomputed using its new key.

Sprites may be single Uint8ClampedArray of pixels (a typical image) or a SpriteMultiple containing multiple Uint8ClampedArray images, keyed by their section. A bush, for example, will have a single sprite; a pipe will have a top sprite and a middle sprite.

To generate your own sprites, you may use ImageReadr.

4.2 Maps

FullScreenMario uses the GameStartr way of storing maps, areas, and locations:

- Maps store a collection of Areas and Locations
- Areas store a setting type (Overworld, Underworld, etc.) and a creation list of commands for creation (next session).
- Locations reference an Area and an x- and y- location in that Area.

FSM.setMap("map", location) may be used to go to a specific map (and, optionally, location number).
FSM.setLocation(location) may be used to go to a location in the current map.

4.2.1 MapsCreatr

Each Area's creation instructions are stored as an Array of Objects. You can see examples of maps in Source/settings/maps.js, and read MapScreenr's readme for a full explanation.
"thing": "Goomba", "x": 32, "y": 8
The coordinate system for creation instructions is not the same as the one used by Things during gameplay. X-distance is still measured from the left, but y-distance is how elevated the Thing is from the floor. This system was implemented to make it more logical to write maps. Coordinates are converted from map to gameplay in FullScreenMario.FullScreenMario.prototype.addPreThing.

4.2.2 MapScreenr

Information on the current visible screen are stored in FSM.MapScreenr. It's the closest thing to a global variable store in FullScreenMario; it stores the offsetX and offsetY of the current screen (from moving to the right). the current map's setting ("Overworld", "Underworld", etc.) and many more, which you can see during gameplay. Like Things, it also stores .top, .right, .bottom, and .left, which are its distance from the starting viewport. If the screen scrolls to the right 50px, for example, FSM.MapScreenr.left will be 50.

4.3 QuadsKeepr

GameStartr uses a grid system for collision detection and bounds checking, run by the QuadsKeepr module. A "Quadrant" refers to a cell in the grid; each Thing knows which Quadrant(s) it overlaps, and each Quadrant knows which Thing(s) overlap it. Collision detection is only calculated between Things that share Quadrants.

Quadrants do not move with MapScreenr, which means Things that aren't moving will stay in the same Quadrants. As MapScreenr scrolls, QuadsKeepr will add or remove Quadrant rows and columns so that the visible screen area is always covered. Adding quadrant rows or cols will trigger MapsHandlr.spawnMap to add PreThings in that area, while removing them triggers MapsHandlr.unspawnMap.

5 Licensing

This project is released and distributed under the **MIT License**

5.1 About the License

The MIT License is a free software license originating at the Massachusetts Institute of Technology (MIT). It is a permissive free software license, meaning that it puts only very limited restriction on reuse and has therefore an excellent license compatibility.The MIT license permits reuse within proprietary software provided all copies of the licensed software include a copy of the MIT License terms and the copyright notice. The MIT license is also compatible with many copyleft licenses, such as the GPL; MIT licensed software can be integrated into GPL software, but not the other way around.