The Internals of the Trails Forward Client

## A high-level summary of the JavaScript client-side systems in Trails Forward

Nick Benson

[nickjbenson@gmail.com](mailto:nickjbenson@gmail.com)

August 30th, 2013

# Impact's JavaScript Object Inheritance

*(based on John Resig's Simple JavaScript Inheritance system)*

Impact needs a module name, a list of required modules, and an object containing variable and function definitions in order to make its class system work.

This reference should give you whatever details you need to know to extend classes using Impact's inheritance system:

<http://impactjs.com/documentation/class-reference/class>

# Updating and Drawing via Impact

Updating and drawing are not threaded in Impact, likely due to the limitations of JavaScript at this point in time. So each 'loop' just gets iterated through in order once per frame. (I'm not certain which is done first, but I suspect it's updating.)

The first thing the update method calls is this.parent(), which calls the parent's update method. For Impact, this means updating any Entities registered with Impact's entity system, and updating any backgroundMaps registered with Impact. Right now, Trails Forward has neither. In fact, this may remain true throughout development, since proprietary entities will likely need to be implemented in order to play nicely with cachedisomaps and draw in the correct order relative to other things on the screen.

The rest of the update method is handling key and mouse input, doing some logistical maintenance on camera positioning (keeping it centered when zooming), ensuring that all assets have been loaded, and calling update() on any other classes that need it, particularly the two different cachedisomaps and the UI. The first is the terrainMap, which just contains water or land tiles, and the second is the featureMap, which contains forest tiles, shoreline tiles, and (in the future when they are implemented) any residential or commercial tile features. The minimap is also loaded if it isn't already, but keep an eye on the minimap class, because it was designed to work best for static, nonchanging data, and that will need to change.

The draw function draws four important things right now. First, it draws the terrainMap, setting up the canvas context, zooming and panning it appropriately before the actual canvas API drawImage call. (drawImage can take javascript <img> objects or predrawn whole canvas objects as arguments for an image source. The cachedisomap algorithm takes advantage of this, using canvas objects as buffers containing pre-drawn ('cached') sections of the map.) Second, it draws non-map images that don't require caching, such as ownership highlighting or tile selection highlighting. Third, it draws the featuremap, including shorelines and trees. Finally, it tells the UI object to draw itself.

# User Interface

The UI class contains update and draw methods of its own. The update method comprises primarily of input handling, calling events on UIElements registered with it based on their positioning relative to the mouse. The draw method basically just cascades the draw call down the tree of UIElements.

The UI object contains a list of registered UIElements. When a mouse event is being considered, the UI will query this list by iterating through it, and UIElements decide whether or not to propogate any requests via function calls to their children. The user interface in TF basically just a tree, with leaf nodes further "up", closer to the screen than parents further behind.

UIElements are very versatile on their own. To produce functionality, one can simply create a UIElement with a rectangle to position it, and then configure it at will. If it needs an image, set the image property. If it needs a click event, assign it a function called onClick. If it needs text, call enableText and pass a function that returns the text to display. To give it a child UIElement, create the child and then pass it to addChild on the parent. (For alignment reasons, a button with text in it will probably consist of a button UIElement with various events specified and a child UIElement consisting only of text.)

Since doing this for extremely common UIElements is annoying, I've created a few subclasses of UIElement that do some things for you. button.js defines a UIElement (Button) with all the functionality a button should have, and can be configured entirely through its constructor. scrollfield.js defines a UIElement with scroll bars on the side and a contentPanel for containing things that need to be scrolled. panel.js defines a UIElement that automatically clips its children to its rectangular region.

This user interface system worked reasonably well for the prototype but it is neither bugless nor the only interface solution. It's possible to overlay pure HTML DOM over Canvas elements, so depending on your preferences, you might want to change how the UI works entirely and take an entirely different route.

# Cached Isometric Maps (cachedisomap.js)

The algorithm for drawing these large maps of data is not entirely my own, as I took a lot of cues from the Isogenic Game Engine and how it handles isometric data maps. Here's my summary of how the algorithm works - it's necessary to understand what this class does, because it will probably need to be modified as the game gets more complicated.

First, we start with all of the data we need to draw the map. If there's no data, the map doesn't freak out; it just doesn't draw any data that is null or undefined. The data is in the form of a two-dimensional array, with each actual datum being another array containing all of the things to draw for a particular tile. (So really it's a three-dimensional array, but that's slightly misleading to say outright.) Position in first two arrays determines where to draw the data, position in the third array determines the drawing order of data for a particular tile.

Data is treated as normal and rectangular for as long as possible. It is only considered "isometric" as soon as it's about to be drawn, at which point the appropriate math is done to convert the data's "real" coordinates to "isometric" coordinates. Here, I use "real" to refer to untransformed, rectangular coordinates, and "isometric" to the transformed coordinates that, when iterated, produces data in an isometric grid.

For each tile, we take its coordinates in real space and convert them to find its coordinates in isometric space on the screen. We then determine which "section" of the map that coordinate lies in. Sections are large square (really square, not isometrically-transformed square) chunks of the map. There are four different resolutions at which we cache each section, at resolution multipliers of 1/1, 1/2, 1/4, and 1/16. Each section also has its own canvas context. A section is considered "cached" when every tile that falls into that section has been drawn onto its canvas context. If a section is not cached, a configurable amount of tiles are drawn onto it every frame until it is cached, at which point the next section is considered. Sections are not drawn to the screen until they are entirely cached at a given resolution. Different resolutions are used depending on the zoom level of the camera, and sections are only drawn if they overlap with the camera's viewport. The update method in cachedisomap will clearly show the priority order on which sections to be caching at any given point in time. Generally, lower resolution sections are cached before higher resolution sections (they take less time), and sections that are on-screen take precedent over sections that are adjacent to the sections that are on-screen.

The result is reminiscent of Google Maps caching, though I haven't the hubris to suggest that the algorithm is nearly as good as the one used by Maps.

# Shoreline and Treeline Generation / Tile Generation Tools

Shorelines should be relatively clear if you look at the getShoreTypes function in main.js in combination with shoreline\_template\_reference\_flat.png, a file that can be found in the public/media folder of the Git repo.

Take a look at the trees\_10\_0 and similar files and see how a system with individual images for each possible permutation of tree tile would be mind-numbingly annoying and terrible. Then look at forest\_tileset\_heavy\_1.png and friends in the same folder and agree that that is far better! The tileset system I have in place for trees is extendable to include residential and commercial features on tiles. The tilesets were generated from the files in the tools folder. When there's a server running, visit /tools/factory.html to see the tree generation in action. Within the tree generator, it's fairly easy to add more variety to the trees that are used. tilefactory.js contains most of the generally-useful tools for generating the tilesets, while vilasfactory.js contains the actual implementations of those tools to generating tilesets for Trails Forward and the Vilas County environment. Residential generation used to exist before tilesets were implemented, but I never got around to changing them to fit the newer tileset (as opposed to individual-tiles) paradigm, hence they're not working right now. A good exercise in understanding the tile factory tools would be to make the Build Residential Tiles button generate a tileset containing some trees and some houses.

# For Next Month and the Future

## Interface:

Investigate what the final interface should look like and decide on a plan of action for producing it. Whether that involves creating a new interface system or extending the one already present is uncertain, and depends on Trails Forward's needs going forward.

## Relatively minor (essentially a list of bugs):

Tile details: They're static right now and should be made dynamic

main.js : this.tileDetailsBox and friends

Owned tiles layer: General aesthetic can be improved, and the minimap should probably also display owned tiles when the layer is turned on.

main.js : this.viewOwnedTilesButton and friends

Panels: There's a clipping problem in the contracts and upgrades window.

Currently selected tile: should also show which tiles are in the megatile that contains the currently selected tile. (Megatiles are 3x3 sets of tiles. Sometimes in the code I refer to a megatile as a 3x3 tile centered on an arbitrary tile, specifically in shoreline code. When talking to the server, however, megatiles are always directly adjacent to one another.)

main.js : draw method, if (this.selectedTile)

Contracts window/other windows: The windows don't refresh if the underlying data used to construct them is changed; right now they only construct themselves once and then just show/hide that window. Obviously they should reflect changes to the underlying data.

main.js : showContractsWindow (and other similarly-named functions all below that point in the file)

Tooltips: Currently don't display the correct information

main.js : generateContractTooltip, generateUpgradeTooltip

Buttons: Make them not activate if there's a leave event before the unclick event

button.js

Minimap: Not optimized for large datasets yet, nor for data that changes after initial loading

isominimap.js