Famous Tutorial: Create an Animated Hamburger Menu Transition - Part 1

Objectives

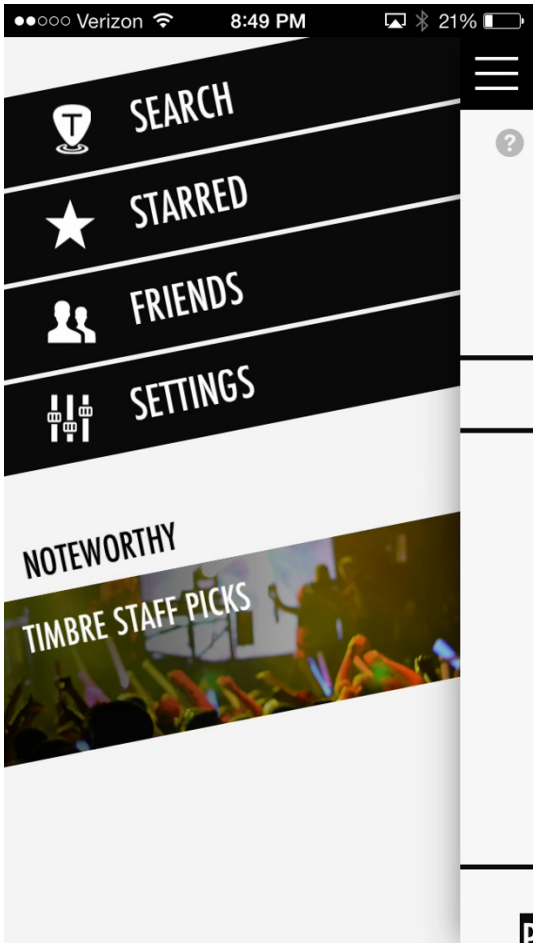
- How to structure a Famous app using nested views
- How to emit and listen for events
- How to create custom animations
- How to capture user input to interact with our app
- How to use Chrome dev tools to emulate mobile devices

Prerequisites

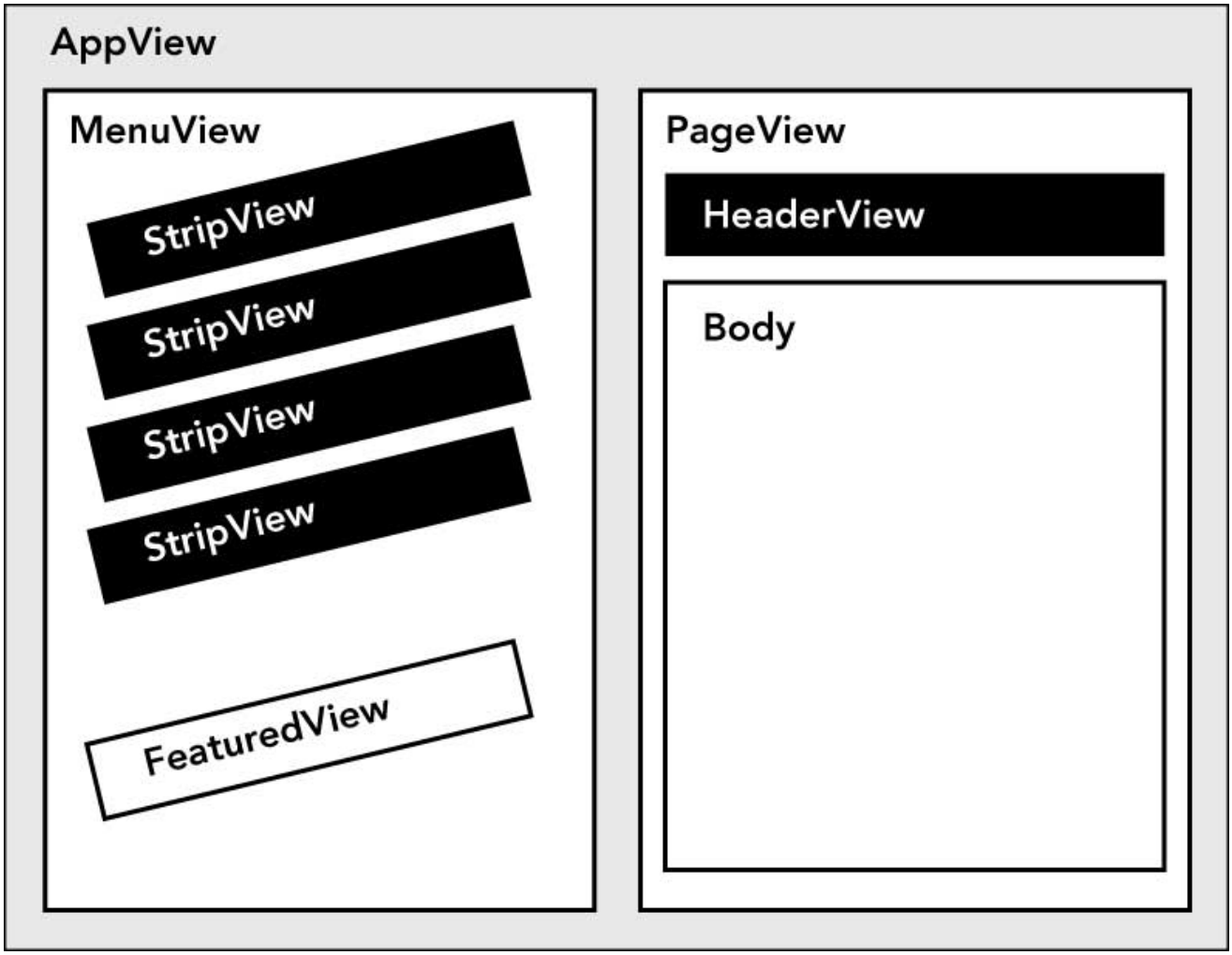
- Finished Walkthrough tutorial
- Intermediate understanding of JavaScript (classes, scope, context)

Design

We're going to recreate the following hamburger menu transition from the Timbre app.



To do so, we're going to organize our app into separate views. **Famous views allow us to build and test our app discretely and animate multiple objects together.** Here's a wireframe of the views that will make up our menu app.



- **AppView** - The app view contains our two main views: menu view and page view. It also contains the logic to slide the page view over the menu view
- **MenuView** - The menu view contains the four strip views and the featured view. It holds the logic to position and animate the strips flying in from the left and the opacitating of the featured view.
- **PageView** - The page view contains the navigation header view and the body where the content of the app would go. For this tutorial, we will not be using real content so the page view will remain very simple.
- **HeaderView** - The header view is the top navigation bar. It contains the hamburger menu button, a search box, and the Timbre logo. When pressed, the hamburger menu button will emit an event that will trigger the page menu to slide. For this tutorial, we will not create a real search box.
- **StripView** - There are actually 4 instances of strip views but all are created from the same StripView class, as we'll see in a second. Each strip view contains an icon and title.
- **FeaturedView** - Simple view with an image

To construct our views, we're going to start with a boilerplate view template. But first, we need to set up our project environment.

Setup

- Clone git@github.com:FamousPrivateBeta/timbre-tutorial-newBase.git
- run `git submodule update --init`
- Check out the `start` branch
- Create a new branch under your name or github handle. Please commit to your branch and not master.
- Create a js/app folder
- Inside app folder, create `MyView.js`

```
define(function(require, exports, module) {
    var Surface         = require('famous/core/Surface');
    var Modifier        = require('famous/core/Modifier');
    var Transform       = require('famous/core/Transform');
    var View            = require('famous/core/View');

    function MyView() {
        View.apply(this, arguments);
    }

    MyView.prototype = Object.create(View.prototype);
    MyView.prototype.constructor = MyView;

    MyView.DEFAULT_OPTIONS = {};

    module.exports = MyView;
});
```

MyView.js (view on github)

We will use `MyView.js` as a starting template for all of our custom views. Our custom views will inherit from the View class. By doing this, we gain the following:

- A render node that all the components in our custom view will attach to
- Event handlers to emit events from
- Options manager to import options

Header View

The first view we're going to display is the header view. However, since the header view is nested inside the page view, which is nested inside the app view, we have to build the last two as well. So, take your `MyView.js` and save them as 3 separate files inside the js/app folder named

- `AppView.js`
- `PageView.js`
- `HeaderView.js`

In each custom view, replace MyView with the appropriate view name (AppView, PageView, HeaderView). What we're doing is creating a separate view class for each of our views. We will then create

new instances of these views and add them to the render tree. Let's do that now.

Connecting Views

In `main.js`, import `AppView`

```
define(function(require, exports, module) {
    var Engine          = require('famous/core/Engine');
    var AppView         = require('app/AppView');

    var mainContext = Engine.createContext();
});
```

Now we create the new instance of our app view and then add it to our context.

```
define(function(require, exports, module) {
    var Engine          = require('famous/core/Engine');
    var AppView         = require('app/AppView');

    var mainContext = Engine.createContext();

    var appView = new AppView();

    mainContext.add(appView);
});
```

main.js (view on github)

Let's now import `PageView` into `AppView` and create a `pageView` instance. Note that the filepath of `PageView` is now referenced from within the js/app folder.

```
define(function(require, exports, module) {
    var Surface         = require('famous/core/Surface');
    var Modifier        = require('famous/core/Modifier');
    var Transform       = require('famous/core/Transform');
    var View            = require('famous/core/View');

    var PageView        = require('./PageView');

    function AppView() {
        View.apply(this, arguments);

        _createPageView.call(this);
    }

    AppView.prototype = Object.create(View.prototype);
```

```
    AppView.prototype.constructor = AppView;

    AppView.DEFAULT_OPTIONS = {};

    function _createPageView() {
        this.pageView = new PageView();
        this.pageMod = new Modifier();

        this._add(this.pageMod).add(this.pageView);
    }

    module.exports = AppView;
});
```

AppView.js (view on github)

A few things to note here:

- We've extracted out the instantiation of pageView into a separate private function just for clarity
- We're creating the pageView as a property on the appView instance by using this.pageView so that we can refer to it elsewehere
- Also, instead of directly adding the pageView to the appView instance, we're adding a modifier in between so that we can transform our pageView
- Lastly, instead of using `.add` and `.add`, when adding to views, we use `._add` and `._link`

We use the same pattern to import and instantiate HeaderView into PageView

```
define(function(require, exports, module) {
    var Surface         = require('famous/core/Surface');
    var Modifier        = require('famous/core/Modifier');
    var Transform       = require('famous/core/Transform');
    var View            = require('famous/core/View');

    var HeaderView      = require('./HeaderView');

    function PageView() {
        View.apply(this, arguments);

        _createHeaderView.call(this);
    }

    PageView.prototype = Object.create(View.prototype);
    PageView.prototype.constructor = PageView;

    function _createHeaderView() {
        this.headerView = new HeaderView();

        this._add(this.headerView);
    }
```

```
    module.exports = PageView;
});
```

PageView.js (view on github)

Great! We've successfully created the scaffold required to display our header view. However if we refresh our browser right now, we won't see anything. That's because there are no renderables (i.e. surfaces) attached to the render tree. Let's go ahead and add some so we can see something on the screen.

In our header view, we are going to create 3 surfaces. One for the hamburger icon, one for the search, and one for the logo. For the purposes of this tutorial, we're just going to place images in all three surfaces. I've already included the images you need in your img folder. Your current folder structure should look something like this

```
timbre-tutorial
├── css
├── img
|   ├── band.png
|   ├── body.png
|   ├── hamburger.png
|   ├── icon.png
|   ├── search.png
|   └── strip-icons
|       ├── friends.png
|       ├── search.png
|       ├── settings.png
|       └── starred.png
├── index.html
├── js
    ├── app
    |   ├── AppView.js
    |   ├── HeaderView.js
    |   ├── MenuView.js
    |   ├── MyView.js
    |   ├── PageView.js
    |   └── StripView.js
    ├── [famous modules]
    └── main.js
```

Our header view is going to be 44px in height and is going to span the entire width of our screen. The first thing we're going to do is create and add a black background surface with these dimensions.

```
define(function(require, exports, module) {
    var Surface         = require('famous/core/Surface');
    var Modifier        = require('famous/core/Modifier');
    var Transform       = require('famous/core/Transform');
    var View            = require('famous/core/View');

    function HeaderView() {
        View.apply(this, arguments);

        _createHeader.call(this);
    }
```

```
    HeaderView.prototype = Object.create(View.prototype);
    HeaderView.prototype.constructor = HeaderView;

    function _createHeader() {
        var backgroundSurface = new Surface({
            size: [undefined, 44],
            properties: {
                backgroundColor: 'black'
            }
        });

        this.hamburgerModifier = new Modifier();

        this._add(backgroundSurface);
    }

    module.exports = HeaderView;
});
```

HeaderView.js

If we now refresh our browser, we should see a black bar across the top of our screen. Next, we're going to add our 3 surfaces with the images.

Inside of our `_createHeader` function, create the following surfaces:

```
        this.hamburgerSurface = new Surface({
            size: [44, 44],
            content: '<img width="44" src="../img/hamburger.png"/>'
        });

        this.searchSurface = new Surface({
            size: [232, 44],
            content: '<img width="232" src="../img/search.png"/>'
        });

        this.iconSurface = new Surface({
            size: [44, 44],
            content: '<img width="44" src="../img/icon.png"/>'
        });
```

HeaderView.js

Create the following modifiers:

```
        this.hamburgerModifier = new Modifier();

        this.searchModifier = new Modifier({
```

```
        origin: [0.5, 0]
    });

    this.iconModifier = new Modifier({
        origin: [1, 0]
    });
```

HeaderView.js

Finally, add the modifiers and corresponding surfaces to the header view.

```
    this._add(this.hamburgerModifier).add(this.hamburgerSurface);
    this._add(this.searchModifier).add(this.searchSurface);
    this._add(this.iconModifier).add(this.iconSurface);
```

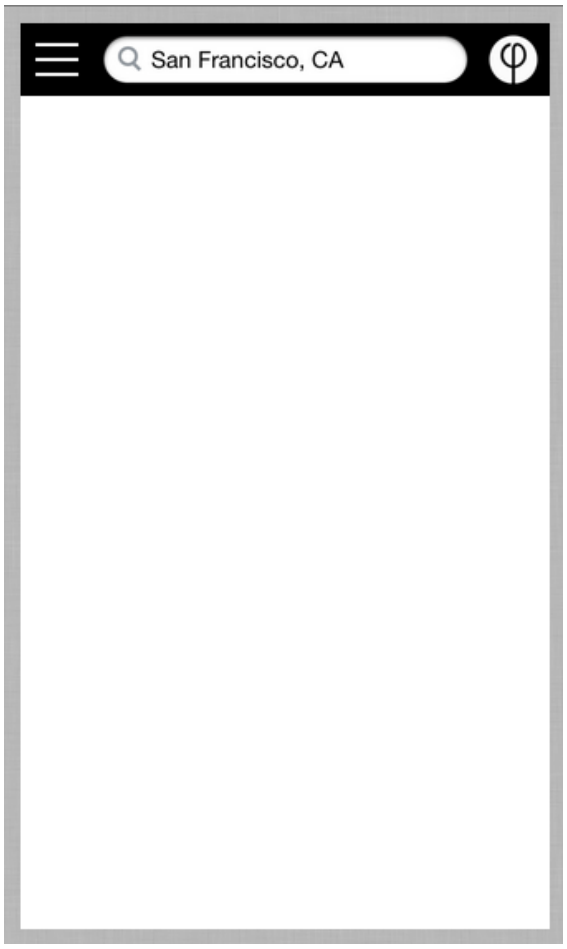
HeaderView.js (view on github)

Things to note:

- The content property of a surface takes html as a string
- The origin property in modifiers specify how the renderables should be positioned relative to its parent. Here, the parent is the full screen. If origin is not specified, it's defaulted to [0, 0] (top-left). Origin [0.5, 0] is top-center and [1, 0] is top-right.
- To use origin, the linked renderable needs to have a .getSize method. The surface class prototype has a getSize method that returns the size specified during declaration.

If you refresh now, you should see the three images placed properly across the top of the screen. Since the original Timbre app was on iOS, the content is scaled to be best viewed in a window 320px wide. For developement, we can use the Emulate feature in Chrome to emulate an iPhone

If you have Chrome 32 installed, bring up Dev Tools then go to Settings/Overrides and then enable Show 'Emulation' view. Then in the Elements tab of Dev Tools, press Esc to bring up the console drawer. You should now have the Emulation tab in the drawer. In the device list, select Apple iPhone 5 and emulate. This will scale your browser window to the correct size, almost. The window is going to be slightly too tall because on a real iPhone 5, the top info bar takes up 20px. So change the resolution of the screen to be 640 x 1096, which scales to 320 x 548 with a device pixel ratio of 2. Now your app should look like the following:



Setting Up Events

Now that we have our header view displaying properly, we're going to emit an event when the hamburger menu button gets tapped. Also, we're going to opacitate out the hamburger menu image when pressed.

Back inside `HeaderView.js`, we're going to create and call another private function `_setListeners`

```
function HeaderView() {
    View.apply(this, arguments);

    _createHeader.call(this);
    _setListeners.call(this);
}
```

HeaderView.js

```
function _setListeners() {
    this.hamburgerSurface.on('touchstart', function() {
        this.hamburgerModifier.setOpacity(0.5);
    }.bind(this));

    this.hamburgerSurface.on('touchend', function() {
        this.hamburgerModifier.setOpacity(1);
```

```
            this._eventOutput.emit('menuToggle');
        }.bind(this));
    }
```

HeaderView.js (view on github)

In _setListeners, we're adding two event listeners for touchstart and touchend on the hamburger surface. On touchstart, we're changing the opacity of the surface to 0.5 and on touchend, changing it back to 1. Also on touchend, we're emitting the `menuToggle` event through `this._eventOutput`, which is an eventHandler inherited from the View class. Since 'this' refers to the header view instance, any events emitted from `this._eventOutput` can be listened to in the parent views of the header view. We will make these connections next. **Important! Make sure to bind your callback functions to the proper context.**

In our app, the immediate parent view of header view is page view. So in `PageView.js`, after we instantiate `this.headerView`, we can add an event listener on `this.headerView` for the `menuToggle` event.

```
    function _createHeaderView() {
        this.headerView = new HeaderView();

        this.headerView.on('menuToggle', function() {
            console.log('toggle');
        });

        this._add(this.headerView);
    }
```

PageView.js (view on github)

If you run this code and click on the hamburger menu button, you should see 'toggle' being logged to your console. That means we're getting touch events. If you don't get the console log, make sure your browser is emulating touch events. Back in your Emulation tab in Dev Tools, go to Sensors and make sure 'Emulate touch screen' is checked.

Okay, now we are listening for the `menuToggle` event in the page view. However we want the entire page view to move when the hamburger surface is clicked, not just the header view. That means we need to bubble up our `menuToggle` event on level higher to app view. One way to do this is the following:

```
    function _createHeaderView() {
        this.headerView = new HeaderView();

        this.headerView.on('menuToggle', function() {
            this._eventOutput.emit('menuToggle');
        }.bind(this));

        this._add(this.headerView);
    }
```

PageView.js

However, there's an easier way to do this by using `pipe`

```
function _createHeaderView() {
    this.headerView = new HeaderView();

    this.headerView.pipe(this._eventOutput);

    this._add(this.headerView);
}
```

PageView.js

The pipe method pipes, or passes along, all the events from its parent object to the argument object. In this case, we're piping the header view to the eventOutput of the page view. This will allows us to listen for the menuToggle event in app view.

While we're still here in `PageView.js`, let's go ahead and create the dummy content for the body.

```
function PageView() {
    View.apply(this, arguments);

    _createHeaderView.call(this);
    _createBody.call(this);
}
```

PageView.js

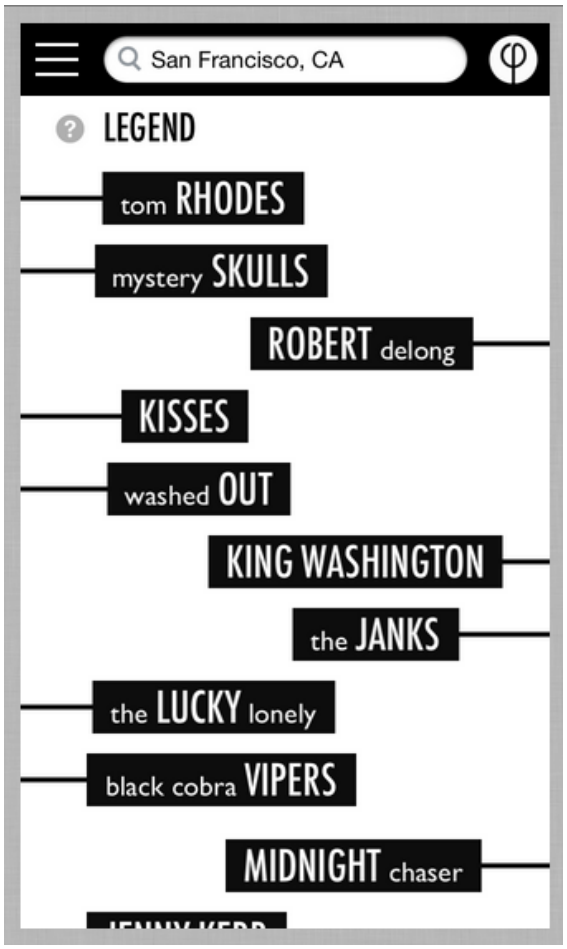
```
function _createBody() {
    this.bodySurf = new Surface({
        size: [320, 504],
        content: '<img width="320" src="../img/body.png"/>'
    });

    this.bodyMod = new Modifier({
        transform: Transform.translate(0, 44, 0)
    });

    this._add(this.bodyMod).add(this.bodySurf);
}
```

PageView.js (view on github)

Our app should now look like this



Alright, now let's get into the fun stuff - animations!

Animating Transitions

In `AppView.js`, we're going to add an event listener for menuToggle emitted from page view. The callback function for this event will be toggleMenu, which will move the page view back and forth over the menu view. In menuToggle, we'll need a flag--this.menuToggle--to be set to determine whether or not the menu has been toggled. We'll set the initial state to be false in `_createPageView`

```
function _createPageView() {
    this.pageView = new PageView();
    this.pageView.on('menuToggle', this.toggleMenu.bind(this));

    this.pageMod = new Modifier();

    this._add(this.pageMod).add(this.pageView);
    this.menuToggle = false;
}

AppView.prototype.toggleMenu = function() {
    if(this.menuToggle) {
        this.slideLeft();
    } else {
        this.slideRight();
```

```
        }
        this.menuToggle = !this.menuToggle;
    };
```

AppView.js

In `toggleMenu`, we're calling one of two methods, `slideLeft` and `slideRight`, depending on the state of `this.menuToggle`. Let's define those now

```
    AppView.prototype.slideLeft = function() {
        this.pageMod.setTransform(Transform.translate(0, 0, 0), {
            duration: 300,
            curve: 'easeOut'
        });
    };

    AppView.prototype.slideRight = function() {
        this.pageMod.setTransform(Transform.translate(276, 0, 0), {
            duration: 300,
            curve: 'easeOut'
        });
    };
```

AppView.js

In both `slideLeft` and `slideRight`, we're using the `setTransform` method of the `pageMod` modifier to translate the page view left and right. The frist argument in setTransform is the matrix transform associated with the translation. The second argument is the transition object which specifies how the translation should be animated. Here we're specifying that the animation should take 300ms to execute and follow the ease in, ease out easing curve.

Since both transition objects are the same, we can do a simple refactor by extracting out the transition object into the default options of our app view. When View.call(this) is run in the instantiation block, the options manager runs and sets `this.options` to `AppView.DEFAULT_OPTIONS` unless the option property is passed in as an argument. If this is confusing, don't worry. We'll get much more practice with options in the next section.

```
    AppView.DEFAULT_OPTIONS = {
        transition: {
            duration: 300,
            curve: 'easeOut'
        }
    };
```

AppView.js

```
    AppView.prototype.slideLeft = function() {
```

```
    this.pageMod.setTransform(Transform.translate(0, 0, 0), this.options.transitio
};

AppView.prototype.slideRight = function() {
    this.pageMod.setTransform(Transform.translate(276, 0, 0), this.options.transit
};
```

AppView.js (view on github)

Now if we click on the hamburger menu button, you should see our page view slide left and right! Cool right? Feel free to play around with the transition object in app view. For the curve, you can use any of the normalized easing curves in famous-animation/Easing, for example:

```
AppView.DEFAULT_OPTIONS = {
    duration: 800,
    curve: Easing.inOutBackNorm
}
```

Just remember to inject the Easing module using require.

```
var Easing = require('famous/transitions/Easing');
```

Part 1 repo on github

Continue to Part 2