Famous Tutorial: Create an Animated Hamburger Menu Transition - Part 2

Review

Thus far we've built about half of our menu transition. When the user clicks or taps on the hamburger menu button, an event is emitted from the header view, piped from the page view to the app view, and then listened to in the app view to toggle the state and animation.

The rest of the tutorial is going to cover the following:

- Building the StripView class
- Passing options between views and the options manager
- Building the menu view with strip view instances
- Animating the strip views
- Integrating user input to swipe the page view

Strip Views

Let's create our first strip view. A strip view contains an icon and a title rotated at an angle with a background surface that is skewed on the end. To add a strip view, we add the strip view to our menu view, which gets added to the app view.

So as before, take your `MyView.js` file and save two copies as `StripView.js` and `MenuView.js`. Inside `AppView.js`, we're going to create and add an instance of `MenuView`.

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

    _createMenuView.call(this);
    _createPageView.call(this);
}
```

```
function _createMenuView() {
    this.menuView = new MenuView();
    this.menuMod = new Modifier({
        transform: Transform.translate(0, 0, -1)
    });

    this._add(this.menuMod).add(this.menuView);
}
```

AppView.js (view on github)

Now in `MenuView.js`, we're going to create an instance of strip view. We're also going to offset the strip view 200px down so that we can see it when it's rotated.

```
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 StripView       = require('./StripView');

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

        _createStripViews.call(this);
    }

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

    MenuView.DEFAULT_OPTIONS = {};

    function _createStripViews() {
        var stripView = new StripView();
        var stripMod = new Modifier({
            transform: Transform.translate(0, 200, 0)
        });

        this._add(stripMod).add(stripView);
    }

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

MenuView.js (view on github)

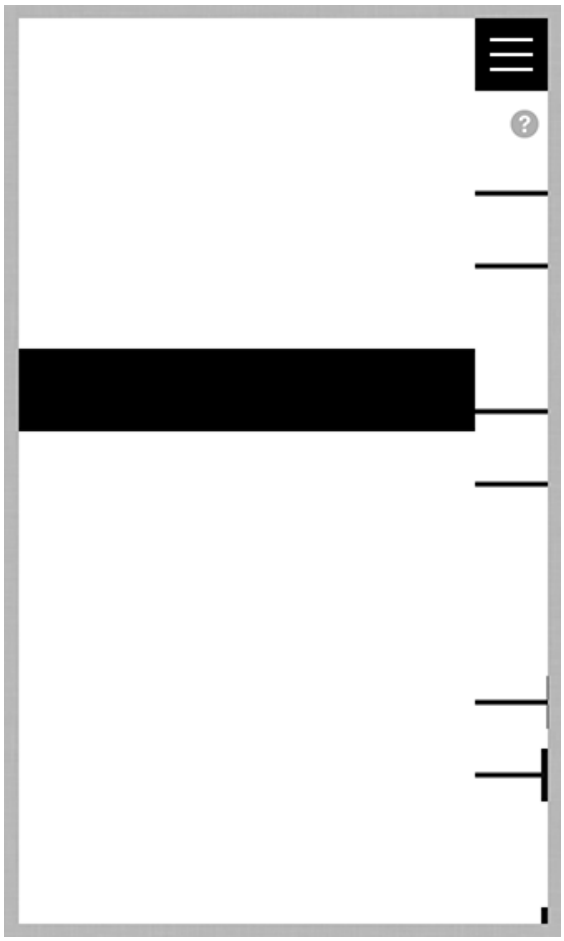
Finally, in `StripView.js`, we're going to create the black backing surface so that we can see something there. We're going to modify the dimensions and placement in the next step.

```
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 StripView() {
        View.apply(this, arguments);
```

```
        _createBacking.call(this);
    }

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

    StripView.DEFAULT_OPTIONS = {};

    function _createBacking() {
        var backSurf = new Surface({
            size: [300, 50],
            properties: {
                backgroundColor: 'black'
            }
        });

        var backMod = new Modifier();

        this._add(backMod).add(backSurf);
    }

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

StripView.js (view on github)

Now when you toggle your menu open, you should see your new strip view.

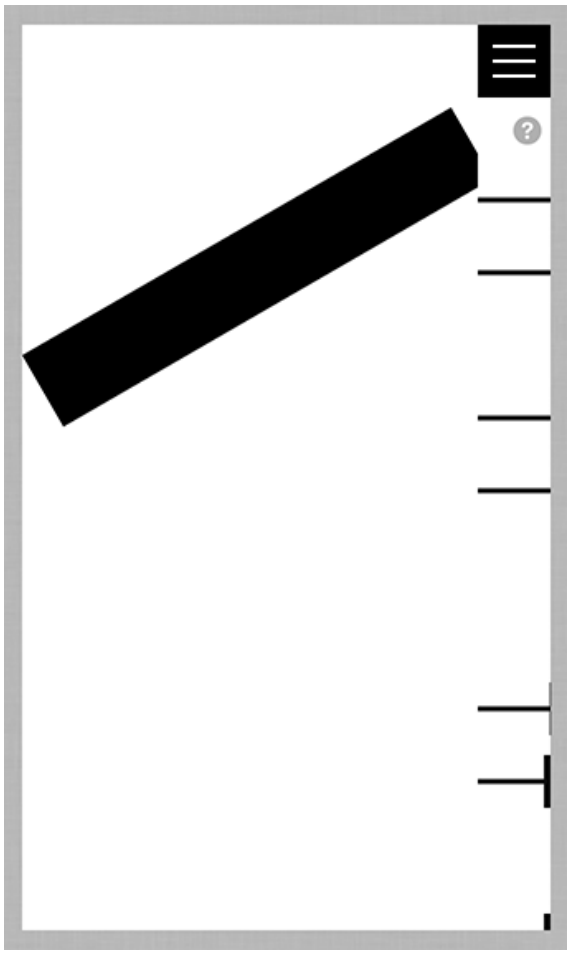


Let's now rotate our backing surface. To do so, we're going to add a transform property in the declaration of `backMod`.

```
function _createBacking() {
    var angle = -Math.PI/6;

    var backSurf = new Surface({
        size: [300, 50],
        properties: {
            backgroundColor: 'black'
        }
    });

    var backMod = new Modifier({
        transform: Transform.rotateZ(angle)
    });

    this._add(backMod).add(backSurf);
}
```

StripView.js (view on github)

You should now see the strip rotated

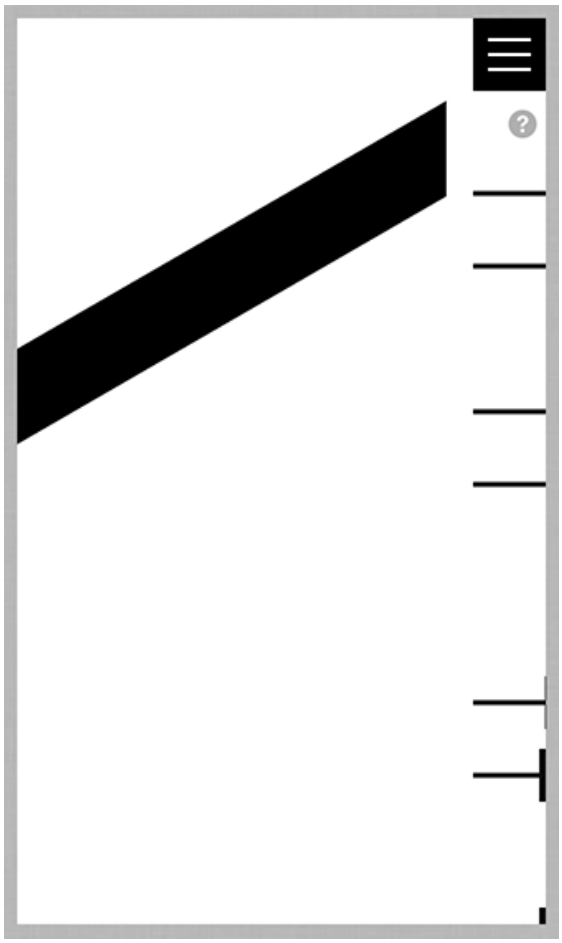


We'll now use the matrix skew function to skew our strip. This will be also applied in our `backMod` transform but it will be used in conjuction with the rotate transform. One way we combine matrix transforms in Famous is to use `Matrix.multiply`.

```
var backMod = new Modifier({
    transform: Transform.multiply(Transform.skew(0, 0, angle), Transform.rotat
});
```

StripView.js (view on github)

And now we see that our strip backing has the desired shape.



Using Options

Currently the strip dimensions and rotation/skew angle are hard coded in `StripView.js`. However when we create multiple strips, we will need to know these values in `MenuView.js`. So to keep these settings in one location, we're going to refactor these options into `MenuView.js` and pass them into `StripView.js`. This paradigm will also enable us to pass parameters such as the icon image url and the strip title for each new strip view.

To do so, we're going to store the keys of the options in `StripView.DEFAULT_OPTIONS` and assign them a default value if we desire.

```
StripView.DEFAULT_OPTIONS = {
    angle: null,
    width: null,
    height: null
};
```

StripView.js (view on github)

When a new `stripView` gets instantiated, the constructor function calls

```
View.apply(this, arguments);
```

which calls the options manager. The options manager sets the `options` property of each `stripView` instance to the default options or, if specified, the options passed into the `StripView` constructor.

So in `MenuView.js`, we'll pass in an options object during instantiation of our `stripView`. This will override the `null` values in `StripView.DEFAULT_OPTIONS`.

```
function _createStripViews() {
    var options = {
        angle: -0.2,
        width: 320,
        height: 54
    };

    var stripView = new StripView(options);

    var stripMod = new Modifier({
        transform: Transform.translate(0, 200, 0)
    });

    this._add(stripMod).add(stripView);
}
```

MenuView.js (view on github)

So now when `_createBacking` is called, `this.options` will contain all the options passed into the constructor plus any default values set in `StripView.DEFAULT_OPTIONS`.

`_createBacking` now looks like this after the refactor

```
function _createBacking() {
    var backSurf = new Surface({
        size: [this.options.width, this.options.height],
        properties: {
            backgroundColor: 'black'
        }
    });

    var backMod = new Modifier({
        transform: Transform.multiply(
            Transform.skew(0, 0, this.options.angle), Transform.rotateZ(this.optio
    });

    this._add(backMod).add(backSurf);
}
```

StripView.js (view on github)

We're going to do one more factor using the options manager to drive this lesson home. In `MenuView.js`,

we've create a local variable `options` insid e the `_createStripViews` function's scope. As you can imagine, when we animate the strips, we're going to need to know these parameters outside of this scope. So we're going to move these values into `MenuView.DEFAULT_OPTIONS`.

```
MenuView.DEFAULT_OPTIONS = {
    angle: -0.2,
    stripWidth: 320,
    stripHeight: 54
};
```

We'll now pass in a different options object during instantiation of `stripView`

```
function _createStripViews() {
    var stripView = new StripView({
        angle: this.options.angle,
        width: this.options.stripWidth,
        height: this.options.stripHeight
    });

    var stripMod = new Modifier({
        transform: Transform.translate(0, 200, 0)
    });

    this._add(stripMod).add(stripView);
}
```

MenuView.js (view on github)

PHEW! Good to be done with that refactor. Now we can start passing in more parameters into `StripView.js` and create multiple strip views with ease.

Multiple Strips

Here's the roadmap for creating multiple strip views

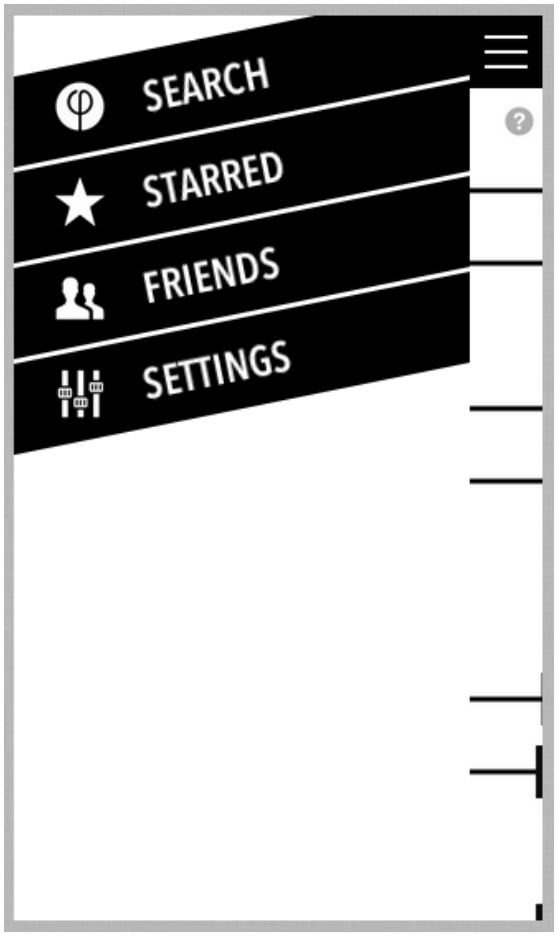
- Title and icon url data will be stored in MenuView and passed into StripView when creating a new strip instance.
- We'll create new surfaces in StripView for the icons and titles
- To position the strips, there will be two components in the y-direction, an offset from the top of the page for all the strips and an incremental offset based on the index of the strip. We'll store these parameters in `MenuView.DEFAULT_OPTIONS`
- To prepare for animating the strips later, we'll store the strip modifers in an array.

All the concepts should be familiar to you already so here's the code for MenuView and StripView.

For styling, we're going to change the font family for the entire app using CSS in `css/app.css`

```
body {
    font-family: 'AvenirNextCondensed-DemiBold';
}
```

Your menu view should now look like this



Touch Input

We now have the basic structure for a hamburger menu style app that reveals the menu on tap. This is good, but we want to allow the user to manually swipe the page view over. To implement this feature, we're going to add a `GenericSync`. We'll create our instance of GenericSync in a new function `_handleTouch` in AppView.

```
function _handleTouch() {
    this.pageViewPos = 0;

    this.sync = new GenericSync(function() {
        return this.pageViewPos;
    }.bind(this), {direction: GenericSync.DIRECTION_X});

    this.pageView.pipe(this.sync);

    this.sync.on('update', function(data) {
        this.pageViewPos = data.p;
```

```
            this.pageMod.setTransform(Transform.translate(data.p, 0, 0));
        }.bind(this));
    }
```

AppView.js (view on github)

When we create our GenericSync, we pass in two arguments. The first is a target function that returns our `pageViewPos`. The second argument is an options object specifying the direction of the sync.

We then add a listener on `update` of the sync to translate our page view. In order for the sync to receive the `update` event, we have to pipe our page view to the sync. If you drag the page view now, you'll see that nothing happens. This is because we never piped from the body surface to the `eventOuput` of page view. Let's do that now.

```
    this.bodySurf.pipe(this._eventOutput);
```

PageView.js (view on github)

Now when you drag the page view around, it will follow your finger/mouse. Then if you click the hamburger menu, it will toggle to the open position. However, if drag the page view again, you'll notice the page view position jumps back to where you had released it after dragging. This is because our variable `this.pageViewPos` never gets updated when the toggle occurs. We could update `this.pageViewPos` in our `slideLeft` and `slideRight` functions, but this is not a good engineering choice. Let me explain.

When we drag the page view half way then toggle the hamburger menu, the page view slides left/right from the position we dragged it to. This means the modifier `pageMod` is keeping track of where the position of the page view is. But because we also track the page view position with our sync, we now have two sources of truth that we have to keep synchronized. This is not ideal and will cause problems.

So, the solution is to get rid of the modifier and track our page view position with only `this.pageViewPos`. However we will need to turn `this.pageViewPos` into a Transitionable object and write a custom render function for AppView.

Transitionables

Transitionable objects are used to store, retrieve, and transition numerical values. The same easing curves and physics transitions used in modifier transitions can be applied to transitionables. In fact, modifiers are built using transitionables.

The refactor for `_handleTouch` is fairly straightforward using the getter and setter methods.

```
    function _handleTouch() {
        this.pageViewPos = new Transitionable(0);

        this.sync = new GenericSync(function() {
            return this.pageViewPos.get(0);
        }.bind(this), {direction: GenericSync.DIRECTION_X});
```

```
        this.pageView.pipe(this.sync);

        this.sync.on('update', function(data) {
            this.pageViewPos.set(data.p);
            this.pageMod.setTransform(Transform.translate(data.p, 0, 0));
        }.bind(this));
    }
```

AppView.js (view on github)

Right now, everything works just as before.

Render Functions

So far, we've been using `.add()` to add renderable elements onto the render tree. Another way to do this is to write a custom render function. Render functions are executed on every animation frame and thus give us much more precise control over the elements in our view.

Render functions return a spec array. Each spec in the spec array contains a renderable object and information on how that object should be displayed. By using a render function, we no longer need `.add`. Let's take a look at the render function and the refactored code.

```
    AppView.prototype.render = function() {
        this.spec = [];

        this.spec.push({
            // opacity: 0.5,
            // size: [300, 300],
            transform: Transform.translate(0, 0, -1),
            target: this.menuView.render()
        });

        this.spec.push({
            transform: Transform.translate(this.pageViewPos.get(), 0, 0),
            target: this.pageView.render()
        });

        return this.spec;
    };
```

AppView.js

Each spec object in the spec array contains a renderable target (make sure to call `.render()`) as well as other properties that define how the renderable should be displayed. For the menu view and page view, we're just going to be setting the transform property but you can also set the opacity and specify the size. It is in the transform of the page view spec that we call `this.pageViewPos.get()` to retrieve the page view position.

Since we're updating the transform here, we no longer need to do it in our sync's on update function.

```
...

this.sync.on('update', function(data) {
    this.pageViewPos.set(data.p);
}.bind(this));

...
```

AppView.js

Lastly, we just need to refactor `slideLeft()` and `slideRight` to set the page position transitionable instead of the modifier transform.

```
AppView.prototype.slideLeft = function() {
    this.pageViewPos.set(0, this.options.transition);
};

AppView.prototype.slideRight = function() {
    this.pageViewPos.set(276, this.options.transition);
};
```

AppView.js Here is the commit for the refactor on github

Now if you drag the page view and toggle, there's no more jump!

Velocity

When the user drags or swipes the page view, it should either slide left or right. To determine which direction to slide the page view, we need to consider both the position and velocity of the page view on the end event of our sync.

Just like we got the position property from `data.p` in update, we can get the velocity from `data.v`.

```
...

this.sync.on('end', (function(data) {
    var velocity = data.v;
    var position = this.pageViewPos.get();

    if(this.pageViewPos.get() > this.options.posThreshold) {
        if(velocity < -this.options.velThreshold) {
            this.slideLeft();
        } else {
            this.slideRight();
        }
    } else {
        if(velocity > this.options.velThreshold) {
```

```
                this.slideRight();
            } else {
                this.slideLeft();
            }
        }
    }).bind(this));

    ...
```

AppView.js

We also need to include our threshold values in our options. **Double check your transition curve.** It should be `'easeOut'`.

```
AppView.DEFAULT_OPTIONS = {
    posThreshold: 138,
    velThreshold: 0.75,
    transition: {
        duration: 300,
        curve: 'easeOut'
    }
};
```

AppView.js

Lastly, after we call `slideLeft` and `slideRight`, we need to update the toggle state. We'll set the flag the in the callback function of the `pageViewPos.set()`.

```
AppView.prototype.slideLeft = function() {
    this.pageViewPos.set(0, this.options.transition, function() {
        this.menuToggle = false;
    }.bind(this));
};

AppView.prototype.slideRight = function() {
    this.pageViewPos.set(276, this.options.transition, function() {
        this.menuToggle = true;
    }.bind(this));
};
```

AppView.js

There's one last detail on the page view controller. The user should not be able to pull the page view left of position zero. We can update this very simply in the sync's on update function.

```
    this.sync.on('update', function(data) {
```

```
            this.pageViewPos.set(Math.max(0, data.p));
        }.bind(this));
```

AppView.js

We've now finished our menu toggle and swipe features. View this code on github

The only thing we have left to do is animate the strips when the page view moves over.

Animating Strips

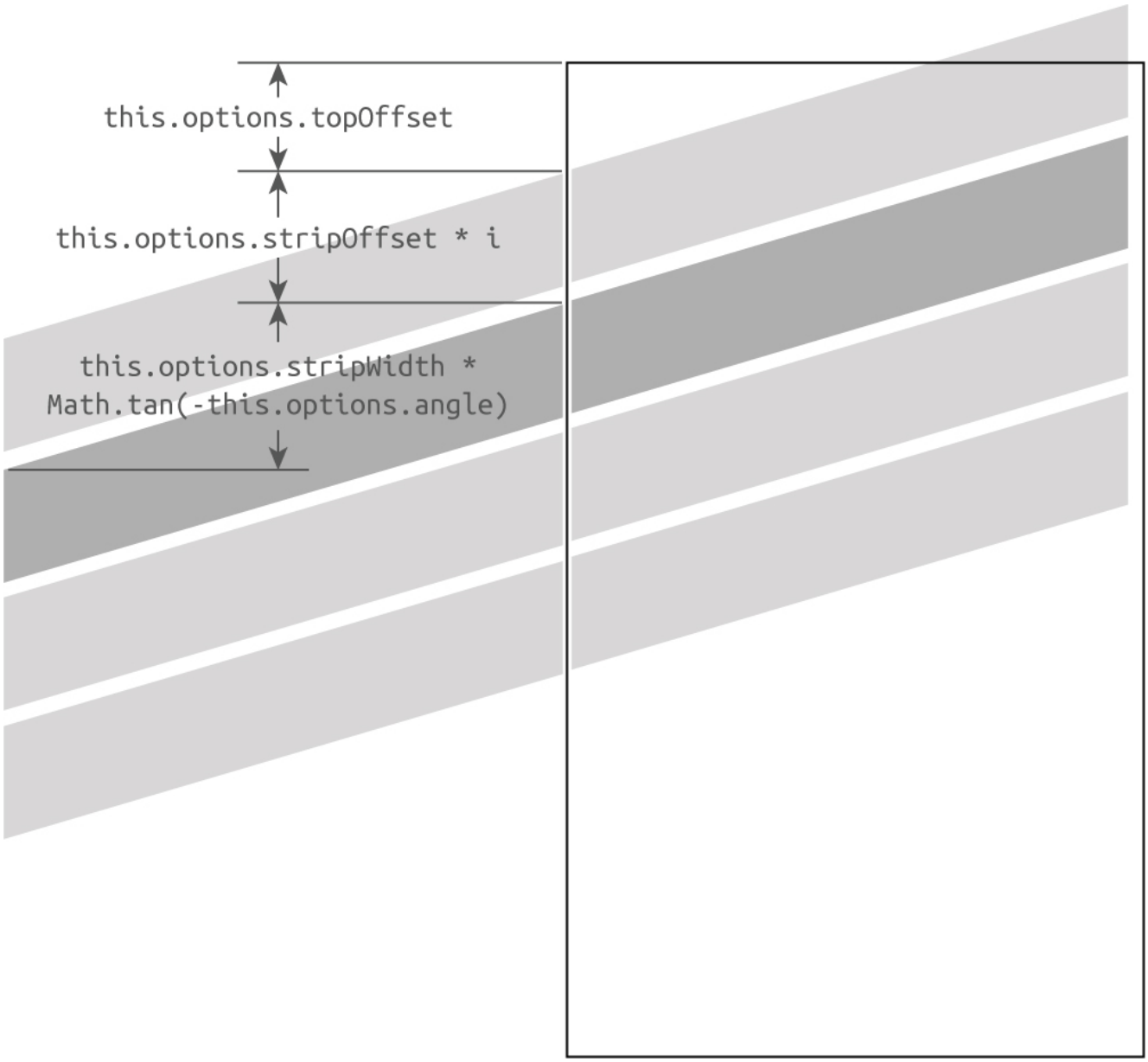
We'll create two methods in the MenuView class called `resetStrips` and `animateStrips`.

`resetStrips` moves the strips to the initial position of the animation (off the left side of the screen).

```
MenuView.prototype.resetStrips = function() {
    for(var i = 0; i < this.stripMods.length; i++) {
        var initX = -this.options.stripWidth;
        var initY = this.options.topOffset
            + this.options.stripOffset*i
            + this.options.stripWidth*Math.tan(-this.options.angle);

        this.stripMods[i].setTransform(Transform.translate(initX, initY, 0));
    }
};

MenuView.prototype.animateStrips = function() {
    this.resetStrips();

    for(var i = 0; i < this.stripMods.length; i++) {
        // use Time.setTimeout instead of window.setTimeout
        // Time can be found in famous-utils

        Time.setTimeout(function(i) {
            var yOffset = this.options.topOffset + this.options.stripOffset * i;

            this.stripMods[i].setTransform(
                Transform.translate( 0, yOffset, 0),
                { duration: this.options.duration, curve: 'easeOut' });
        }.bind(this, i), i*this.options.staggerDelay);
    }
};
```

MenuView.js

Here is a diagram showing how the initial and final positions for the animation are calculated



We also need to specify the `stripOffset`, `duration`, and `staggerDelay` parameters in the menu view options

```
MenuView.DEFAULT_OPTIONS = {
    angle: -Math.atan(74/370),
    stripWidth: 320,
    stripHeight: 54,
    topOffset: 37,
    stripOffset: 58,
    duration: 400,
    staggerDelay: 35
};
```

MenuView.js (view on github)

In AppView, we will trigger the animation in two places

1. on calling `slideRight()` by toggling the menu
2. when `pageViewPos` is zero and the user moves the page right

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

```
    ...

    this.sync.on('update', function(data) {
        if(this.pageViewPos.get() === 0 && data.p > 0) {
            this.menuView.animateStrips();
        }

        this.pageViewPos.set(Math.max(0, data.p));
    }.bind(this));

    ...
```

AppView.js (view on github)

Finishing Touches

We're pretty much done with our menu transition. We have one view left to add--the featured view--but there's really nothing new concepts there. You should try to build the last transition yourself. All the images are in the assets folder. If you need a hint, you can check the commit.

The last detail we can add is a shadow on the left side of our page view. To do this, we'll just create a backing surface that's the full size of our device window and apply a box shadow. Here is the commit.

There you have it folks! Now you know a great deal about how to build in Famous. Until next time.

Completed project repo on github

Back to Part 1