UIScrollView's Inertia, Bouncing and Rubber-Banding with UIKit Dynamics

06 Jul 2014

Two months ago Ole Begemann wrote a great article about rebuilding `UIScrollView` from scratch: Understanding UIScrollView. A couple of days later Rounak Jain and Grant Paul added inertial scrolling, bouncing and rubber-banding to it using Facebook's Pop framework. Since then, I've wanted to recreate the same effect with UIKit Dynamics and have finally done that this week.

Animating bounds with UIKit Dynamics

Objects animatable with UIKit Dynamics have to conform to the `UIDynamicItem` protocol:

```
@protocol UIDynamicItem <NSObject>

@property (nonatomic, readwrite) CGPoint center;
@property (nonatomic, readonly) CGRect bounds;
@property (nonatomic, readwrite) CGAffineTransform transform;

@end
```

Dynamics uses `center` and `transform` properties to move items based on its internal algorithm[1], and `bounds` property to compute collisions. It follows that we can't use Dynamics directly on the scroll view to animate its `bounds`. `UIDynamicItem` is a protocol, though, so we can create a plain old `NSObject` subclass conforming to it:

```
@interface CSCDynamicItem : NSObject <UIDynamicItem>

@property (nonatomic, readwrite) CGPoint center;
@property (nonatomic, readonly) CGRect bounds;
@property (nonatomic, readwrite) CGAffineTransform transform;

@end

@implementation CSCDynamicItem
- (instancetype)init {
    self = [super init];

    if (self) {
        // Sets non-zero `bounds`, because otherwise Dynamics throws an exception.
        _bounds = CGRectMake(0, 0, 1, 1);
    }


    return self;
```

```
}
@end
```

and use this class to drive changes of the scroll view's `bounds`. For the simplicity's sake we'll assume that the dynamic item's `center` maps to the scroll view's `bounds.origin`. There is a couple of ways to bind these values:

1. We could register ourselves as an observer of the dynamic item's `center` key path and update the `bounds` when its value changes.
2. We could pass the scroll view instance to the dynamic item and update its bounds from within `setCenter:`.
3. We could leverage the fact that `UIDynamicBehavior` contains an `action` property, described as:

> *The block you want to execute during dynamic animation. The dynamic animator calls the action block on every animation step.*

We'll go with the last one, because it's the most succinct and—I think—it fits this case the best. Here it is in code:

```
behavior.action = ^{
    CGRect bounds = weakSelf.bounds;
    bounds.origin = weakSelf.dynamicItem.center;
    weakSelf.bounds = bounds;
};
```

Inertial Scrolling

We're now ready to add an inertial scrolling. There are two things to it: 1) when a user lifts a finger off the screen after panning, the scroll view should continue to scroll with the same velocity vector and 2) the scrolling should slow down with time. While browsing through the documentation of built-in behaviors we quickly notice this one sentence in `UIDynamicItemBehavior` description:

> *One notable and common use of a dynamic item behavior is to confer a velocity to a dynamic item to match the ending velocity of a user gesture.*

The class is really flexible, it supports both linear and angular motions. We're going to use `-addLinearVelocity:forItem:` to *push* the dynamic item with a velocity grabbed from the gesture recognizer. Adding inertia is a simple matter of changing `resistance` property's value. Here's a full setup of this behavior:

```
self.dynamicItem.center = self.bounds.origin;
UIDynamicItemBehavior *decelerationBehavior = [[UIDynamicItemBehavior alloc] initWithItems:(
[decelerationBehavior addLinearVelocity:velocity forItem:self.dynamicItem];
decelerationBehavior.resistance = 2.0;

__weak typeof(self) weakSelf = self;
decelerationBehavior.action = ^{
    CGRect bounds = weakSelf.bounds;
    bounds.origin = weakSelf.dynamicItem.center;
    weakSelf.bounds = bounds;
```

```
};

[self.animator addBehavior:decelerationBehavior];
```

You can also see the whole file at this stage on GitHub.

Rubber-Banding

Rubber-banding is the easiest part and it actually doesn't use Dynamics at all. We just have to alter the bounds during panning according to the equation:

```
f(x, d, c) = (x * d * c) / (d + c * x)

where,
x – distance from the edge
c – constant (UIScrollView uses 0.55)
d – dimension, either width or height
```

Here's how it works:

Bouncing

We'll use `UIAttachmentBehavior` for a bouncing effect. The idea here is simple: when the `bounds.origin` crosses the visible area (derived from `contentSize`) we calculate the anchor, which is a final position the user should end up at:

```
//  mostly based on Grant Paul's code from the Pop-based version
CGPoint maxBoundsOrigin = CGPointMake(self.contentSize.width - bounds.size.width,
                                      self.contentSize.height - bounds.size.height);
CGPoint target = bounds.origin;
if (outsideBoundsMinimum) {
    target.x = fmin(maxBoundsOrigin.x, fmax(target.x, 0.0));
    target.y = fmin(maxBoundsOrigin.y, fmax(target.y, 0.0));
} else if (outsideBoundsMaximum) {
    target.x = fmax(0, fmin(target.x, maxBoundsOrigin.x));
    target.y = fmax(0, fmin(target.y, maxBoundsOrigin.y));
}
```

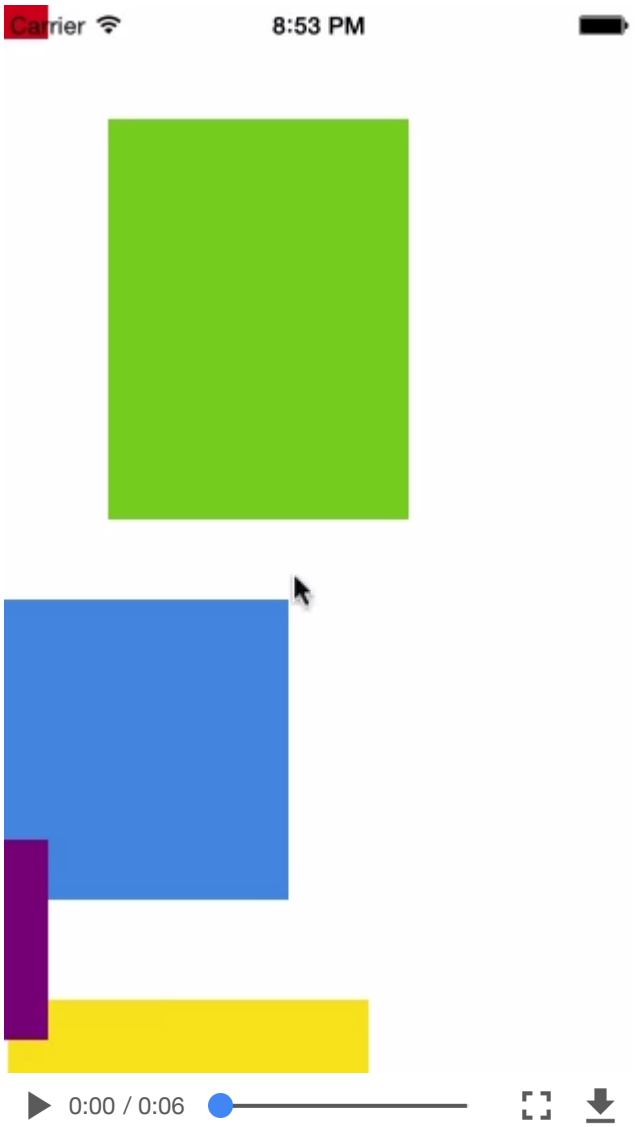
and attach the behavior to it:

```
UIAttachmentBehavior *springBehavior = [[UIAttachmentBehavior alloc] initWithItem:self.dynar
// Has to be equal to zero, because otherwise the bounds.origin wouldn't exactly match the t
springBehavior.length = 0;
// These two values were chosen by trial and error.
springBehavior.damping = 1;
springBehavior.frequency = 2;

[self.animator addBehavior:springBehavior];
```

The rest of the method is mostly boilerplate, you can see it on GitHub.

Everything looks great until we try the scroll view with scrolling enabled in both directions. If a panning gesture has non-zero vertical and horizontal velocities, there's an ugly oscillation when the `bounds.origin` is close to the `target`:



It happens, because the attachment behavior doesn't simply simulate spring animation along the line. It can be influenced by other behaviors, and it is in our case, by `decelerationBahavior`, causing it to rotate around its anchor point. We could try to remove `decelerationBahavior` while adding `springBehavior`, but it would in turn zero out the velocity.

After some debugging[2] I noticed that the problem lies in the calculation of the spring's anchor. Animations are discrete, so for example when the scroll view is pushed to the left, `bounds.origin` can take the following values:

```
x                y
46.984947   78.164795
36.891747   82.781387
20.600927   90.232750
8.031227    95.982079
-4.141042   101.549622
-13.288508  105.733643
```

`x` is never exactly equal to 0, so we have to calculate `y` for `x = 0` manually by solving a system of two linear equations:

```
y_1 = a*x_1 + b
y_2 = a*x_2 + b
```

With that, we're able to calculate the point at which the `bounds.origin` crossed the left edge and attach the spring to the correct position. Calculations are analogous for other edges. Here is a final version in action:

Conclusion

We learned how to use Dynamics in less common situations and how to leverage a flexibility provided by its protocol-based design. The final result is really close to that provided by UIKit. I have to admit, though, that the code is less obvious than when done with Pop, because:

1. We had to use an intermediary object to animate the `bounds`. It's currently—and I doubt it ever will be—not possible to create relationships between custom properties with Dynamics.
2. `UIAttachmentBehavior`, unlike Pop's `POPSpringAnimation`, doesn't have a `velocity` property, so we had to keep `decelerationBehavior` and calculate the anchor point manually.

The final version is available on GitHub.

Sidenote: The work on Pop started long before Dynamics was introduced. We can only wonder what would have happened with Pop, had Dynamics been introduced earlier.

---

1. It uses Box2D, a 2-dimensional physics simulator engine, under the hood. ↩
2. In the meantime I made a quick fix by adding the `UICollisionBehavior`. Here's how it worked. It's an interesting effect, but not what we wanted to achieve. ↩

I'm Arek Holko, an iOS engineer in Warsaw, Poland. Follow me on Twitter and subscribe to my newsletter to stay updated. Learn more about me.

Recent Articles:

- Optimizing Swift build times
- Surprising behavior of non-optional @NSManaged properties
- Catching Leaky View Controllers Without Instruments
- Avoiding Third-Party UI Libraries