Extracted from:

RubyMotion

iOS Development with Ruby

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

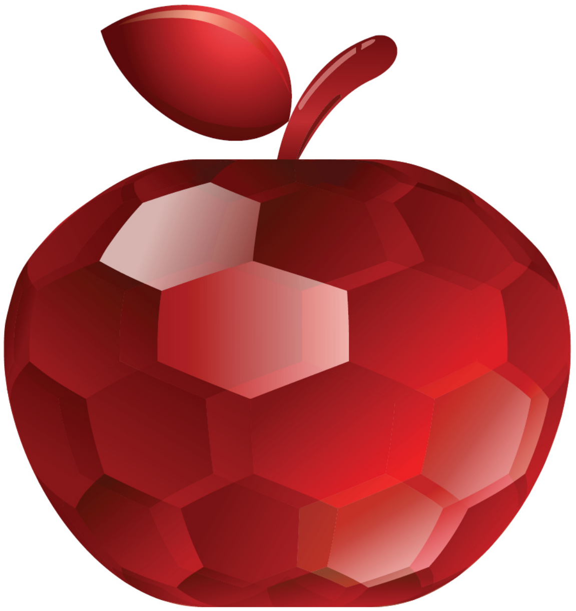
The Pragmatic Programmers

Pragmatic exPress

RubyMotion

iOS Development with Ruby

Updated for RubyMotion 2



Clay Allsopp

Foreword by Laurent Sansonetti, lead developer of RubyMotion

edited by Fahmida Y. Rashid

RubyMotion

iOS Development with Ruby

Clay Allsopp

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Fahmida Y. Rashid (editor)
Kim Wimpsett (copyeditor)
David J. Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-28-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P3.0—July 2014

Animating Views

We talked about how important views are on iOS, but apps are also known for their smooth animations. Perhaps one reason slick transitions and niceties are so pervasive in iOS apps is that they're so darn easy to implement. And we're going to add that to our little box app right now.

Let's add another button, Remove, which will fade out the most recently added view and slide all others to new positions in its place. That might sound complicated, and on other platforms or frameworks it might be, but the iOS animation APIs make it painless. All we do is tell the system what properties of our views to animate and how long that animation should take.

We will add yet another button to our AppDelegate and wire its target/action callbacks for removing a view.

views/Boxey_animations/app/app_delegate.rb

```ruby
@add_button.addTarget(
  self, action:"add_tapped", forControlEvents:UIControlEventTouchUpInside)
@remove_button = UIButton.buttonWithType(UIButtonTypeSystem)
@remove_button.setTitle("Remove", forState:UIControlStateNormal)
@remove_button.sizeToFit
@remove_button.frame = CGRect.new(
  [@add_button.frame.origin.x + @add_button.frame.size.width + 10,
    @add_button.frame.origin.y],
  @remove_button.frame.size)
@window.addSubview(@remove_button)
@remove_button.addTarget(
  self, action:"remove_tapped",
  forControlEvents:UIControlEventTouchUpInside)
```

Pretty easy, right? Basically, all we did was set a frame and add a subview; that's nothing new. Now we need to implement that remove_tapped() callback. It's going to be longer than our add_tapped() method, so we'll take it slow. First, we need to find the objects we're interested in.

views/Boxey_animations/app/app_delegate.rb

```ruby
def remove_tapped
  other_views = @window.subviews.reject { |view|
    view.is_a?(UIButton)
  }
  last_view = other_views.last
  return unless last_view && other_views.count > 1
```

Because our buttons are also subviews of the window, we need to prune them and make sure we deal only with the blue boxes. There are better ways to architect this (such as storing the boxes in some independent array), but this works with what we have. Next, we do the actual animations!

views/Boxey_animations/app/app_delegate.rb

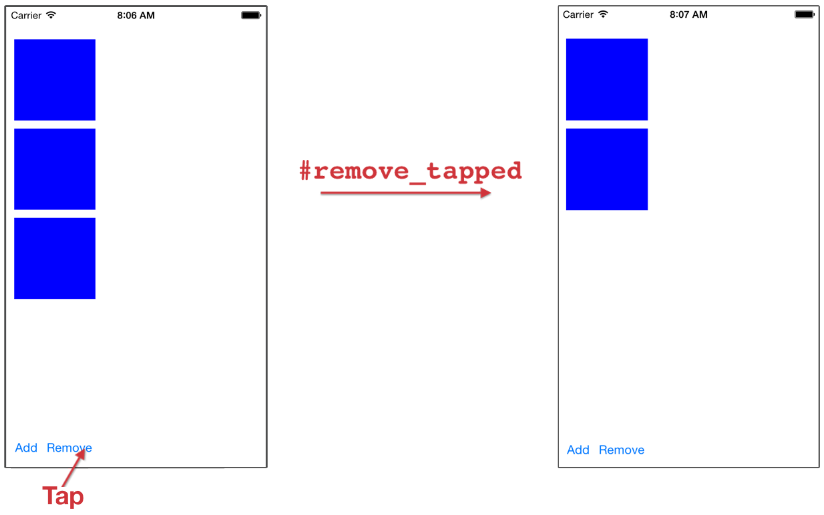
```
  animations_block = lambda {
    last_view.alpha = 0
    last_view.backgroundColor = UIColor.redColor
    other_views.reject { |view|
      view == last_view
    }.each { |view|
      new_origin = [
        view.frame.origin.x,
        view.frame.origin.y - (last_view.frame.size.height + 10)
      ]
      view.frame = CGRect.new(
        new_origin,
        view.frame.size)
    }
  }
  completion_block = lambda { |finished|
    last_view.removeFromSuperview
  }
  UIView.animateWithDuration(0.5,
    animations: animations_block,
    completion: completion_block)
end
```

Animations revolve around the UIView.animateWithDuration:animations: group of methods (you can also use animateWithDuration:delay:options:animations:completions: if you need to fine-tune things). Any alterations to your views made in the lambda we pass for animations will animate if possible. Beyond the basics like frame and opacity, most sensible properties of UIView will work as expected.[3] In our case, we fade out the box by setting the floating-point alpha to zero. Then we enumerate through all the other views and adjust their frames.

We use the optional completion: argument to get a callback when the animation finishes. This block takes one boolean argument, which tells us if the callback has been fired when the animation actually completed (the callback may fire prematurely if the animation has been canceled elsewhere). This is a good place to clean up our views, which we do here by invoking removeFromSuperview(). This will remove the view from its parent's subviews and be erased from the screen.

The animation function looks a bit strange because of the multiple lambdas, but it's no different from changing those properties of a view when they're static. Give it a rake, add some boxes, and then watch them float away with the Remove button:

3. You can find a full list of animatable properties at: http://developer.apple.com/library/ios/#documentation/uikit/reference/uiview_class/uiview/uiview.html.



Adding Static Text with UILabel

Animations are fun, but we also need to display information long enough for the user to actually read it. In most cases, we can use UILabel to display static text. Labels can be very flexible, allowing you to change everything from the font to minute adjustments with the text baseline, and are really easy to get up and running. Let's add one to our little app.

We're going to add a UILabel to each box, displaying its index in subviews. We probably wouldn't ship that sort of feature, but it's really handy for debugging and might give us a better idea of what's going on in our animation. UILabel is really lightweight, so it won't be a pain to add.

Adding labels is going to occur in a new method called add_label_to_box(). This method will figure out the index of a given box's UIView instance and add the correct UILabel. This is the important part of the code, so let's take a look at it first.

views/Boxey_label/app/app_delegate.rb

```
def add_label_to_box(box)
  box.subviews.each do |subview|
    subview.removeFromSuperview
  end

  index_of_box = @window.subviews.index(box)
  label = UILabel.alloc.initWithFrame(CGRectZero)
  label.text = "#{index_of_box}"
  label.textColor = UIColor.whiteColor
  label.backgroundColor = UIColor.clearColor
  label.sizeToFit
  label.center = [box.frame.size.width / 2, box.frame.size.height / 2]
  box.addSubview(label)
end
```

We start by removing all subviews from box, which handles the case where we call this method multiple times on the same view (which we will). Our label is initialized with CGRectZero, which is shorthand for a rectangle at the origin and no size. After we set the text appropriately, we call sizeToFit() just like UIButton. The UILabel implementation of sizeToFit() will precisely fill the frame to fit the text, leaving no padding. Then we use the center property of UIView, which is shorthand for putting the center of a view at a point (as opposed to the upper-left corner).

Remember how we said subviews are positioned within their parent? Even though we set the label to be centered at a coordinate like (50, 50), it can exist at a different point within the window. As our animation slides the box, its label will move too.

Not too bad, right? The only UILabel-exclusive properties in this example are text and textColor; everything else is inherited from UIView. Now we need to actually call this method.

We usually want to go through all the boxes each time we update the labels so we can be absolutely sure our labels are in sync with subviews. To make our lives easier, we're going to refactor the logic for picking out boxes from @window.subviews into one method that simply returns only the boxes.

views/Boxey_label/app/app_delegate.rb
```
def boxes
  @window.subviews.reject do |view|
    view.is_a?(UIButton) or view.is_a?(UILabel)
  end
end
```

We can combine our two new methods into one really great helper method that takes care of everything.

views/Boxey_label/app/app_delegate.rb
```
def add_labels_to_boxes
  self.boxes.each do |box|
    add_label_to_box(box)
  end
end
```

Finally, we can put these to some use, first in application:didFinishLaunchingWithOptions:

views/Boxey_label/app/app_delegate.rb
```
  @window.addSubview(@blue_view)
➤ add_labels_to_boxes
```

and then down in add_tapped().

views/Boxey_label/app/app_delegate.rb

```
@window.insertSubview(new_view, atIndex:0)
➤ add_labels_to_boxes
```

Lastly, we need to reset the labels for each box after we run the removal animation. We're going to change our other_views to use self.boxes instead of its own array construction. Then we're going to use our handy add_labels_to_boxes() to sync all the labels again.

views/Boxey_label/app/app_delegate.rb

```
def remove_tapped
➤   other_views = self.boxes
    last_view = other_views.last
```

views/Boxey_label/app/app_delegate.rb

```
completion_block = lambda { |finished|
  last_view.removeFromSuperview
➤   add_labels_to_boxes
}
```

Whew. Our UILabel was only a small part of our changes, but now we can clearly see how our view hierarchy behaves at runtime. Run the app, and you should see labels appear as in the following figure.

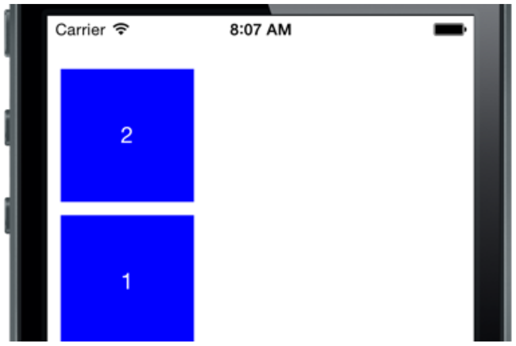


Figure 4—Labels reset for each box

Making Text Dynamic with UITextField

Most apps have more than just buttons and labels; usually we need the user to enter some data, like a tweet or email address. UITextField is the basic view we use to grab string-type input, and now we're going to add one to our little box app. It will let the user pick the UIColor of our boxes using simple commands like "red" and "blue." Let's get started.

First we need to add the field to the view hierarchy. UITextField puts many configuration options at our disposal, from fonts to the look of the Return key. We won't be using all of its properties today, but you can consult Apple's documentation on the class for more information.[4] Our field should be added in app_delegate.rb like so:

views/Boxey_textfield/app/app_delegate.rb

```
@remove_button.addTarget(
  self, action:"remove_tapped",
  forControlEvents:UIControlEventTouchUpInside)

@color_field = UITextField.alloc.initWithFrame(CGRectZero)
@color_field.borderStyle = UITextBorderStyleRoundedRect
@color_field.text = "Blue"
@color_field.enablesReturnKeyAutomatically = true
@color_field.returnKeyType = UIReturnKeyDone
@color_field.autocapitalizationType = UITextAutocapitalizationTypeNone
@color_field.sizeToFit
@color_field.frame = CGRect.new(
  [@blue_view.frame.origin.x + @blue_view.frame.size.width + 10,
    @blue_view.frame.origin.y + @color_field.frame.size.height],
  @color_field.frame.size)
@window.addSubview(@color_field)

➤ @color_field.delegate = self
```

Like every other view in the window, we spend some time setting up the frame and positioning it exactly where we want it. returnKeyType() and similar properties control exactly what they say; the only cryptic property we use is UITextBorderStyleRoundedRect, which adds a nice border and inner shadow to our field. By default, UITextFields have empty backgrounds and no default styling.

The most important part of our addition is setting @color_field's delegate. Much like our application uses AppDelegate as its delegate, other objects use the delegation pattern as a way of sending callback events. The UITextFieldDelegate specification lists all of the methods the delegate object can implement.[5] We aren't required to implement any of them, but we will implement textFieldShouldReturn: to intercept when the Return/Done key is pressed.

views/Boxey_textfield/app/app_delegate.rb

```
def textFieldShouldReturn(textField)
  color_tapped
  textField.resignFirstResponder
```

4. You can find the UITextField reference at http://developer.apple.com/library/ios/#documentation/uikit/reference/UITextField_Class/Reference/UITextField.html.
5. You can find the UITextFieldDelegate reference at http://developer.apple.com/library/ios/#Documentation/UIKit/Reference/UITextFieldDelegate_Protocol/UITextFieldDelegate/UITextFieldDelegate.html.

```
  false
end
```

resignFirstResponder looks a little cryptic, but in reality it simply hides the keyboard. In iOS, there's a concept of a *responder chain* that determines how events such as taps are propagated among our objects. We won't deal with the responder chain in this book, but the important thing to remember is that the first responder of a text field is almost always the virtual keyboard. You also may notice that we explicitly return false from textFieldShouldReturn:, but why is that? Whatever you return from this method decides whether the UITextField carries out the default behavior of its Return key; in our case, we're hiding the keyboard, and the normal action should be avoided.

Finally, we need to implement the color_tapped() method we called in textFieldShould-Return:. We're going to read the text property of the text field and use the Ruby metaprogramming send() method to create a UIColor from that string.

views/Boxey_textfield/app/app_delegate.rb

```
def color_tapped
  color_prefix = @color_field.text
  color_method = "#{color_prefix.downcase}Color"
  if UIColor.respond_to?(color_method)
    @box_color = UIColor.send(color_method)
    self.boxes.each do |box|
      box.backgroundColor = @box_color
    end
  else
    UIAlertView.alloc.initWithTitle("Invalid Color",
        message: "#{color_prefix} is not a valid color",
        delegate: nil,
        cancelButtonTitle: "OK",
        otherButtonTitles: nil).show
  end
end
```

Since we're feeling friendly, we alert the user if there is no such UIColor for their input. But if we do succeed in creating a color object, we assign it to a @box_color instance variable. We need to go back to other parts of the code to make sure they also use @box_color; that way, events like adding a new box work as expected.

views/Boxey_textfield/app/app_delegate.rb

```
  @window.makeKeyAndVisible
➤ @box_color = UIColor.blueColor
  @blue_view = UIView.alloc.initWithFrame(CGRect.new([10, 40], [100, 100]))
➤ @blue_view.backgroundColor = @box_color
  @window.addSubview(@blue_view)
```

views/Boxey_textfield/app/app_delegate.rb

```
def add_tapped
```

```
  new_view = UIView.alloc.initWithFrame(CGRect.new([0, 0], [100, 100]))
➤ new_view.backgroundColor = @box_color
```

All we're doing here is changing the hard-coded use of blueColor to our new instance variable. And there's one more thing: we need to fix our boxes() method to ignore the new UITextField.

views/Boxey_textfield/app/app_delegate.rb

```
def boxes
  @window.subviews.reject do |view|
➤   view.is_a?(UIButton) or view.is_a?(UILabel) or view.is_a?(UITextField)
  end
end
```

Fantastic; let's run rake again and play with the text field (see *Figure 5, Playing with the text field*, on page 12). Be sure to try more exotic colors like "cyan" and "magenta," too.

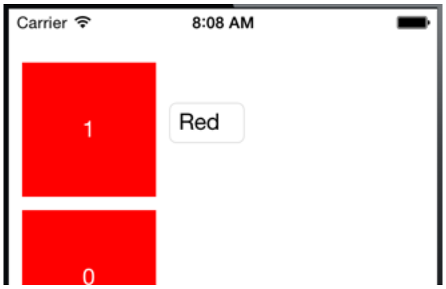


Figure 5—Playing with the text field

Exploring RubyMotion Libraries

We whipped up a pretty interesting app using a relatively small amount of code; however, we also used some vestigial Objective-C patterns that look obviously out of place. This is one area where the RubyMotion community is stepping up and wrapping un-Ruby code into more idiomatic structures. Several libraries and RubyGems[6] are available that could have helped us manage our views.

For example, Sugarcube (https://github.com/rubymotion/sugarcube) would have allowed us to replace those long animation method names with very concise functions such as fade_out() and move_to().

```
last_view.fade_out { |view|
```

6. For a full explanation of RubyGems and RubyMotion, check out *Third-Party Libraries and RubyMotion*, on page ?.

```
  last_view.removeFromSuperview
}

other_views.each do |view|
  new_origin = [
    view.frame.origin.x,
    view.frame.origin.y - (last_view.frame.size.height)
  ]

  view.move_to new_origin
end
```

Much better, right? And for more complex apps, the Teacup library[7] allows you to construct views using CSS-esque style sheets. Our blue boxes might have Teacup style sheets defined like this:

7. https://github.com/rubymotion/teacup

```
Teacup::Stylesheet.new :app do
  style :blue_box,
    backgroundColor: UIColor.blueColor,
    width: 100,
    height: 100
end
```

Third-party libraries like these are helping RubyMotion become more than just an Objective-C/Ruby mashup. As you can see in the previous examples, they can dramatically change how we express what we are trying to accomplish in code. Later in *Representing Data with Models* and *Example: Writing an API-Driven App*, we'll actually use some third-party RubyMotion frameworks to simplify otherwise complex elements of our apps.

But even without those niceties, we've gone from an empty app to an interactive, animated UI in the span of a few quick examples. There are far more included UIView subclasses than we have time for, but the ones we've covered should make those a cinch to learn when the time comes.

By making a slightly more ambitious app, we have also gotten a chance to see how the Ruby language can make our iOS development lives a little easier. Take my word for it, RubyMotion methods such as Array#select and string formatting ("#{ruby_code_here}") are more concise than their Objective-C counterparts. Then again, we still have some very non-Ruby practices that leave much to be desired, such as UITextField's delegate pattern.

In the course of working on views, our AppDelegate got pretty crowded with all kinds of code: helper functions, button callbacks, view creation…the works. Real apps have much more robust organization in the form of controllers, which we'll cover now in *Organizing Apps with Controllers*.