Pete Callaway

Blog (/) About (/about)

iOS8 presentation controllers

29 Jun 2014

iOS 8 adds a new class, `UIPresentationController`, that works along-side the classes and protocols introduced in iOS 7 to help us implement custom view controller transitions. I always find it easier to understand a new API by working through an example, so let's build the following transition

The Xcode project for this article is available on GitHub (https://github.com/PeteC/PresentationControllers).

Implementing a custom transition

There are two objects that we need to implement for our custom transition. A subclass of `UIPresentationController` and a class that implements the `UIViewControllerAnimatedTransitioning` protocol.

Our `UIPresentationController` subclass will be responsible for animating views that aren't part of either the view controller being presented or the controller that's doing the presentation. In our case, this will be the semi-transparent red view.

Our `UIViewControllerAnimatedTransitioning` class will be responsible for animating the presented view controller

Let's look at the `UIPresentationController` first

UIPresentationController

There are five methods we need to override in our `UIPresentationController` subclass

- `presentationTransitionWillBegin`
- `presentationTransitionDidEnd:`
- `dismissalTransitionWillBegin`
- `dismissalTransitionDidEnd:`
- `frameOfPresentedViewInContainerView`

`presentationTransitionWillBegin` is called whenever the presentation is about to begin. This is where we'll add our red dimming view to the hierarchy and setup an animation of it's alpha from 0 to 1.

```
override func presentationTransitionWillBegin() {
    // Add the dimming view and the presented view to the heirarchy
    self.dimmingView.frame = self.containerView.bounds
    self.dimminView.alpha = 0.0

    self.containerView.addSubview(self.dimmingView)
    self.containerView.addSubview(self.presentedView())

    // Fade in the dimming view alongside the transition
    let transitionCoordinator = self.presentingViewController.transitionCoordinator()
    transitionCoordinator.animateAlongsideTransition({(context: UIViewControllerTransitionCoordinatorContext!) -> Void in
        self.dimmingView.alpha  = 1.0
    }, completion:nil)
}
```

By using the the presenting controller's `UIViewControllerTransitionCoordinator`, we make sure our animation runs along side any other animations.

`presentationTransitionDidEnd:` is called whenever our presenting transition has ended and provides a bool to indicate whether the transition completed. In our example, we can use this to remove the dimming view if our presentation didn't complete.

```
override func presentationTransitionDidEnd(completed: Bool)  {
    // If the presentation didn't complete, remove the dimming view
    if !completed {
        self.dimmingView.removeFromSuperview()
    }
}
```

This covers showing our red dimming view, we now need to animate it out and remove it when our presentation is dismissed. As you'd expect, `dismissalTransitionWillBegin` is where we can animate the alpha of our our red dimming view back to 0.

```
override func dismissalTransitionWillBegin()  {
    // Fade out the dimming view alongside the transition
    let transitionCoordinator = self.presentingViewController.transitionCoordinator()
    transitionCoordinator.animateAlongsideTransition({(context: UIViewControllerTransitionCoordinatorContext!) -> Void in
        self.dimmingView.alpha  = 0.0
    }, completion:nil)
}
```

We've now animated the alpha of our dimming view to 0, but we also need to remove it when our presentation has been completly dismissed. For this we override `dismissalTransitionDidEnd:`.

```
override func dismissalTransitionDidEnd(completed: Bool) {
    // If the dismissal completed, remove the dimming view
    if completed {
        self.dimmingView.removeFromSuperview()
    }
}
```

There's one last method we need to override in our `UIPresentationController` subclass. In our custom presentation, the presented view controller doesn't fit the whole screen. It's the `UIPresentationViewController`'s responsibility to define the final frame of the presented view controller. We do this in the `frameOfPresentedViewInContainerView` method

```
override func frameOfPresentedViewInContainerView() -> CGRect {
    // We don't want the presented view to fill the whole container view, so inset it's frame
    var frame = self.containerView.bounds;
    frame = CGRectInset(frame, 50.0, 50.0)

    return frame
}
```

The completed class can be viewed here (https://github.com/PeteC/PresentationControllers/blob/master/App/PresentationControllers/CustomPresentationController.swift).

UIViewControllerAnimatedTransitioning

I covered the `UIViewControllerAnimatedTransitioning` protocol in a previous blog post (/blog/2013/09/29/interactive-transitions/). With the introduction of `UIPresentationController`, our `UIViewControllerAnimatedTransitioning` class has less to do as it is now only responsible for animating the view controller views and not any additional views, such as our red dimming view.

There are two protocol methods we need implement

- `transitionDuration:`
- `animateTransition:`

We'll also add a class property that we can set on initialisation that indicates whether our class should present or dismiss the view controller.

The simplest method to override is `animatedTransition:`. All we need to do is return the animation duration for our presentation.

In `animateTransition:` we animate the presented view controller into or out of view.

```
func animateTransition(transitionContext: UIViewControllerContextTransitioning!)  {
    if isPresenting {
        animatePresentationWithTransitionContext(transitionContext)
    }
    else {
        animateDismissalWithTransitionContext(transitionContext)
    }
}
```

```
func animatePresentationWithTransitionContext(transitionContext: UIViewControllerContextTransitioning) {
    let presentedController = transitionContext.viewControllerForKey(UITransitionContextToViewControllerKey)
    let presentedControllerView = transitionContext.viewForKey(UITransitionContextToViewKey)!
    let containerView = transitionContext.containerView()!

    // Position the presented view off the top of the container view
    presentedControllerView.frame = transitionContext.finalFrameForViewController(presentedController)
    presentedControllerView.center.y -= containerView.bounds.size.height

    containerView.addSubview(presentedControllerView)

    // Animate the presented view to it's final position
    UIView.animateWithDuration(transitionDuration(transitionContext), delay: 0.0, usingSpringWithDamping: 1.0, initialSpringVelocity: 0.0, options: .AllowUs
        presentedControllerView.center.y += containerView.bounds.size.height
    }, completion: {(completed: Bool) -> Void in
        transitionContext.completeTransition(completed)
    })
}

func animateDismissalWithTransitionContext(transitionContext: UIViewControllerContextTransitioning) {
    let presentedControllerView = transitionContext.viewForKey(UITransitionContextFromViewKey)!
    let containerView = transitionContext.containerView()!

    // Animate the presented view off the bottom of the view
    UIView.animateWithDuration(transitionDuration(transitionContext), delay: 0.0, usingSpringWithDamping: 1.0, initialSpringVelocity: 0.0, options: .AllowUs
        presentedControllerView.center.y += containerView.bounds.size.height
    }, completion: {(completed: Bool) -> Void in
            transitionContext.completeTransition(completed)
    })
}
```

The completed class can be viewed here (https://github.com/PeteC/PresentationControllers/blob/master/App/PresentationControllers/CustomPresentationAnimationController.swift).

Using the custom presentation classes

Now we've implemented the classes we need for our custom presentation, let's see how we can use them. There are a number of ways to get our classes to be used, but a simple way is for our presented view controller to assign itself as it's own `UIViewControllerTransitioningDelegate`.

```
init(coder aDecoder: NSCoder!) {
    super.init(coder: aDecoder)
    self.commonInit()
}

init(nibName nibNameOrNil: String!, bundle nibBundleOrNil: NSBundle!)  {
    super.init(nibName: nibNameOrNil, bundle: nibBundleOrNil)
    self.commonInit()
}

func commonInit() {
    self.modalPresentationStyle = .Custom
    self.transitioningDelegate = self
}
```

Our view controller can now provide an instance of our `UIPresentationController` class whenever it's presented...

```
func presentationControllerForPresentedViewController(presented: UIViewController!, presentingViewController presenting: UIViewController!, sourceViewContro
    if presented == self {
        return CustomPresentationController(presentingViewController: presenting, presentedViewController: presented)
    }
    else {
        return nil
    }
}
```

...and an instance of our `UIViewControllerAnimatedTransitioning` class to animate the presentation or dismissal

```
func animationControllerForPresentedController(presented: UIViewController!, presentingController presenting: UIViewController!, sourceController source: UI
    if presented == self {
        return CustomPresentationAnimationController(isPresenting: true)
    }
    else {
        return nil
    }
}

func animationControllerForDismissedController(dismissed: UIViewController!) -> UIViewControllerAnimatedTransitioning! {
    if dismissed == self {
        return CustomPresentationAnimationController(isPresenting: false)
    }
    else {
        return nil
    }
}
```

The completed class can be viewed here (https://github.com/PeteC/PresentationControllers/blob/master/App/PresentationControllers/MessageViewController.swift).

Summary

Like my previous post (/blog/2013/09/29/interactive-transitions/), hopefully this will help you to get started creating your own custom presentations. The sample Xcode project is available on GitHub (https://github.com/PeteC/PresentationControllers).