DEVCON

RWDevCon 2018 Vault

Table of Contents: Overview

Table of Contents: Extended

14: Custom Views

Learn three different ways of creating and manipulating custom views. First, learn how to supercharge your IB through code and create unique views using storyboards. Next, dive into creating exible and reusable views. Finally, bring it all together with some Core Graphics and Core Animation pizazz!

39 Custom Views: Demo 1

By Lea Marolt Sonnenschein

In this demo, you will learn how to incorporate generics and POP (protocol oriented programming) to your UITableView setup to reduce code bloat and write code that is more dynamic and reusable.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

Note: Begin work with the starter project in **Demo1\starter**.

1) Add an extension to UITableView

In **UITableView+Extensions.swift** add this below `import UIKit`:

```
extension UITableView {
  func registerCell<Cell: UITableViewCell>(_ cellClass: Cell.Type) {
    register(cellClass, forCellReuseIdentifier: String(describing:
cellClass))
  }

  func dequeueReusableCell<Cell: UITableViewCell>(forIndexPath indexPath:
IndexPath) -> Cell {
    let identifier = String(describing: Cell.self)
    guard let cell = self.dequeueReusableCell(withIdentifier: identifier,
for: indexPath) as? Cell else {
      fatalError("Error for cell id: \(identifier) at \(indexPath)")
    }
    return cell
  }
}
```

2) Add a generic table view cell

In **GenericTableViewCell.swift** replace the implementation with:

```
class GenericTableViewCell<View: UIView>: UITableViewCell {

  var cellView: View? {
    didSet {
      guard cellView != nil else { return }
      setUpViews()
    }
  }

  override init(style: UITableViewCellStyle, reuseIdentifier: String?) {
    super.init(style: .default, reuseIdentifier: reuseIdentifier)
    selectionStyle = .none
  }

  required init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
  }

  private func setUpViews() {
    guard let cellView = cellView else { return }

    addSubview(cellView)
    cellView.pinEdgesToSuperview()
  }
}
```

3) Refactor the tableView setup in TodayViewController

In **TodayViewController.swift** in `setUpViews()` replace this line

```
tableView.register(CardViewCell.self, forCellReuseIdentifier:
String(describing: CardViewCell.self))
```

with

```
tableView.registerCell(GenericTableViewCell<CardView>.self)
```

Next, in `tableView(_:cellForRowAt:)` replace this line

```
guard let cardCell = tableView.dequeueReusableCell(withIdentifier:
String(describing: CardViewCell.self), for: indexPath) as? CardViewCell
else { return UITableViewCell() }
```

with:

```
let cardCell = tableView.dequeueReusableCell(forIndexPath: indexPath) as
GenericTableViewCell<CardView>
```

Replace this line:

```
guard let cardView = cardCell.cardView else {
```

with:

```
guard let cardView = cardCell.cellView else {
```

And replace this line:

```
cardCell.cardView = cardView
```

with this:

```
cardCell.cellView = cardView
```

4) That's it!

Congrats, at this time you should have a good understanding of how to create more flexible, reusable and smart custom views by supercharging your UITableViews and UITableViewCells with generics.

Custom Views: Demo 2

By Lea Marolt Sonnenschein

In this demo, you will learn how to create custom presentation transitions and animate like a pro. You will create the transition from our TodayViewController to the DetailViewController just like the Today tab does in the AppStore app and you will lay out the ground work for the dismissal animation.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

Note: Begin work with the starter project in **Demo2\starter**.

1) Implement a custom modal presentation manager

In **TodayViewController.swift**, declare a new variable, `transitionManager`, at the top of the class definition:

```
private let transitionManager = TransitionManager()
```

Scroll down to `tableView(_ tableView: didSelectRowAt:)`, and before you present `detailViewController`, add these two lines:

```
detailViewController.transitioningDelegate = transitionManager
detailViewController.modalPresentationStyle = .overFullScreen
```

Build and run to make sure you can transition to and from the detail view.

2) Copy the selected cardView for further

animation manipulation

Add the following above the `if isPresenting` test, right after we create the `cardView` property:

```
let cardViewCopy = createCardViewCopy(cardView: cardView)
```

`createCardViewCopy(cardView:)` grabs the selected `cardView` and makes a copy of it for you to manipulate.

3) Add cardViewCopy to your containerView

Add the following to the block of code that's executed when `isPresenting` is true, just before the call to `transitionContext.completeTransition(true)`:

```
// Add the transition cardView
cardView.isHidden = true
containerView.addSubview(cardViewCopy)
```

This hides the original `cardView` and adds the `cardViewCopy` to the `containerView`.

Build and run to make sure you're on track. At this point, you should see your `cardViewCopy` on top of the `toVC`'s `view`.

4) Move and scale the cardViewCopy

When you move from `TodayViewController` to `DetailViewController` you have to de-round the corners of the `cardViewCopy` and pin it to the top. Only after that's done, can you complete the transition.

To do this, complete the private function stub of `moveAndConvertCardView(cardView: containerView: yOriginToMoveTo: completion:)`:

```
private func moveAndConvertCardView(cardView: CardView,
containerView: UIView, yOriginToMoveTo: CGFloat, completion: @escaping ()
->()) {
  // 1: Layout the current cardViewCopy and update its constraints
  cardView.layoutIfNeeded()
  cardView.updateConstraints(for: .full)

  UIView.animate(withDuration: 0.6, delay: 0.0,
    usingSpringWithDamping: 1.0, initialSpringVelocity: 0.5,
    options: .curveEaseOut, animations: {

    // 2: Animate corner rounding, move y origin to 0
    cardView.containerView.layer.cornerRadius = 0
    cardView.frame.origin.y = 0
```

```
      // 3: Animate the constraint changes
      cardView.layoutIfNeeded()

    }, completion: { (completed) in
      completion()
    })
  }
```

This method accepts a completion block where you can call `transitionContext.completeTransition(true)` after all the animations are over.

Now go back to `animateTransition(using:)` and replace the call to `transitionContext.completeTransition(true)` in the block that executes if `isPresenting` is true, with:

```
// Complete transition
moveAndConvertCardView(cardView: cardViewCopy, containerView:
containerView, yOriginToMoveTo: 0, completion: {
  transitionContext.completeTransition(true)
})
```

Build and run now and you should be able to see the card that you select animate to the top and fill the entire width of the screen.

5) Create a seamless transition

You might be pretty frustrated at this point, because after you animate your `cardViewCopy` you're stuck in a weird limbo. The presentation was technically completed, but you can't dismiss it. That's because the `cardViewCopy` is presented on top of the `toVC` view.

To make this transition look seamless, you have to first hide the `toVC`'s `view`, then unhide it in the animation completion and remove the `cardViewCopy` from the `containerView`.

In `animateTransition(using:)` at the top of the `if isPresenting` block change the `viewsAreHidden` variable to true:

```
toVC.viewsAreHidden = true
```

Now, at the end of the animation, unhide the `toVC`'s views, removed the `cardViewCopy` and show the original `cardView` in the completion block so that it looks like this:

```
// Complete transition
moveAndConvertCardView(cardView: cardViewCopy, containerView:
containerView, yOriginToMoveTo: 0, completion: {
  toVC.viewsAreHidden = false
  cardViewCopy.removeFromSuperview()
```

```
    cardView.isHidden = false
    transitionContext.completeTransition(true)
  })
```

Build and run to make sure you can present and dismiss successfully and that the card animation looks correct now.

6) Add some bounce to your animation

The card animation looks ok, but it's a little lifeless. You can fix that by giving the `cardViewCopy` a more granular `CASpringAnimation` to move its y origin. The `yOriginMoveAndBounceAnimation` function does exactly that.

In `moveAndConvertCardView(cardView: containerView: yOriginToMoveTo: completion:)`, before you enter the `UIView` animation block, create a new animation property:

```
  let yOriginAnimation = yOriginMoveAndBounceAnimation()
```

Now, inside the animation block, replace `cardView.frame.origin.y = 0` with:

```
  yOriginAnimation.toValue = 0
  cardView.layer.add(yOriginAnimation, forKey: "yMove")
```

This will animate the frame instead of manually moving it.

Build and run now and you'll notice that the animation doesn't quite work as expected. The `cardViewCopy` frame doesn't actually move. That's because the `CAAnimation` is working on the lower-level `layer` property, so we need to explicitly tell it where to move.

Luckily, you know the absolute position of the `cardViewCopy` and can use it's `origin.y` property to know just how much to move.

7) Pass the absolute origin.y to your animation

In `moveAndConvertCardView(cardView: containerView: yOriginToMoveTo: completion:)` change the `toValue` for the `yOriginAnimation` in the `UIView` animation block from 0 to `-yOriginToMoveTo`.

```
  yOriginAnimation.toValue = -yOriginToMoveTo
```

Now add the new correct value in your presentation call:

```
moveAndConvertCardView(cardView: cardViewCopy, containerView:
containerView,
yOriginToMoveTo: cardViewCopy.frame.origin.y, completion: {
  toVC.viewsAreHidden = false
  cardViewCopy.removeFromSuperview()
  cardView.isHidden = false
  transitionContext.completeTransition(true)
})
```

Build and run now and you should see the `cardViewCopy` moving to the correct position, this time with some spring in its step!

8) Animate the bottom

The top of the animation is good to go, but the bottom is still lacking. The AppStore animates a view in a scroll like fashion to reveal the text in the `DetailViewController` in this case, so let's try to achieve that effect.

The `TransitionManager` already has a property called `whiteScrollView` that you can use. At the beginning, the `whiteScrollView` needs to have the same `position`, `size` and `cornerRadius` as the `cardView`.

In `animateTransition(using:)`, in the `if isPresenting` block, add the following just before you add the `cardViewCopy`:

```
// Configure and insert whiteScrollView view to animate with the cardView
whiteScrollView.frame = cardView.containerView.frame
whiteScrollView.layer.cornerRadius = cardView.layer.cornerRadius
cardViewCopy.insertSubview(whiteScrollView, aboveSubview:
  cardViewCopy.shadowView)
```

Now go to `moveAndConvertCardView(cardView: containerView: yOriginToMoveTo: completion:)` and add this right before the last call to `cardView.layoutIfNeeded()` in the `UIView` animation block:

```
self.whiteScrollView.layer.cornerRadius = 0
self.whiteScrollView.frame = containerView.frame
```

Build and run now and marvel at your creation!

9) That's it!

Congrats, at this time you should have a good understanding of creating custom modal presentation and using basic and more complex animations to achieve smooth and seamless transitions!

Custom Views: Demo 3

By Lea Marolt Sonnenschein

In this demo, you will implement a second method to dismiss the DetailViewController when the user scrolls down on it.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

Note: Begin work with the starter project in **Demo3/starter**.

1) Create an image from your views

Head over to **UIView+Extensions.swift** and add the following:

```
extension UIView {
  func createImage() -> UIImage? {
    UIGraphicsBeginImageContextWithOptions(frame.size, false, 0)
    drawHierarchy(in: frame, afterScreenUpdates: true)

    let image = UIGraphicsGetImageFromCurrentImageContext()

    UIGraphicsEndImageContext()

    return image
  }
}
```

This adds an `extension` to `UIView` that creates a `UIImage` from the specified rectangle of a view.

2) Create a snapshot view of the currently

visible screen parts

You can use your new `createImage()` function to create a `UIImage` of the `DetailViewController`'s view.

Go to **DetailViewController.swift**. There's a new `UIImageView` property there called `snapshotView` that will hold the image. Find the `createSnapshoOfView` function and add these two lines where it says `// Add Image here:`:

```
let snapshotImage = view.createImage()
snapshotView.image = snapshotImage
```

The `createSnapshoOfView` function creates a snapshot of the current view, adds a shadow behind it, and a blur effect behind the entire view to better distinguish the snapshot from the background.

Now, add a call to this function at the end of `viewDidLoad()`:

```
createSnapshoOfView()
```

This adds the new `snapshotView` to the `view` of the `DetailViewController` but hides it initially so it doesn't disturb the user's flow.

3) Allow scrolling to decide which view to present

When the user scrolls down, you want to make it appear as if the currently presented screen they're seeing is shrinking (whilst rounding its corners). Equally, when the user is scrolling up, you want to make the view transform back into the original view. If the user scrolls down far enough, the view should dismiss.

As the `DetailViewController` is the the `UIScrollViewDelegate` delegate for the `scrollView`, we can easily tap into where and when the user scrolls. Scroll down to the bottom of the file, and inside your new `scrollViewDidScroll(_:)` function you will find these two properties.

```
let yPositionForDismissal: CGFloat = 20
let yContentOffset = scrollView.contentOffset.y
```

`yPositionForDismissal` is the boundry point for when to dismiss the view. You'll use `yContentOffset` soon.

Now add the following:

```
if yContentOffset < 0 {
  viewsAreHidden = true
```

```
    snapshotView.isHidden = false
  } else {
    viewsAreHidden = false
    snapshotView.isHidden = true
  }
```

If `yContentOffset` is `< 0` you know that the user scrolled down and you want to show your snapshot, otherwise, you want to show the original view.

4) Use the scrolling to create scale and rounding animations

You're correctly switching between the snapshot view and the real view, but now it's time to animate. You'll use the `yContentOffset` constant to calculate how much to transform the scale of your `snapshotView`.

In the `if yContentOffset < 0` code block, add the following at the end:

```
let scale = (100 + yContentOffset)/100
snapshotView.transform = CGAffineTransform(scaleX: scale, y: scale)
```

This defines the required scale to use to transform the `snapshotView`.

Add this right after the transforms:

```
snapshotView.layer.cornerRadius = -yContentOffset > 20 ?
  20 : -yContentOffset
```

This uses the `yContentOffset` to change the `cornerRadius` property of the layers for `snapshotView`.

Finally, add this:

```
if yPositionForDismissal + yContentOffset <= 0 {
  close()
}
```

If the user has scrolled past the threshold for dismissal set in `yPositionForDismissal`, dismiss the view by calling `close()`.

Build and run to see that everything builds correctly and check out your new animation. It's looking a little janky, isn't it? The dismissal is a little too sudden and the snapshot seems to move a little oddly. You'll fix that in the next step.

5) Spruce it up

You're almost there, but not quite. To make the dismissal feel a little more natural to the user, you should add a bit of delay to it, and make sure that the `scrollView` can't move past that point.

In `scrollViewDidScroll(_:)` function find this block of code:

```
if yPositionForDismissal + yContentOffset <= 0 {
  close()
}
```

and replace it with:

```
if yPositionForDismissal + yContentOffset <= 0 {
  scrollView.setContentOffset(CGPoint(x: 0, y: -yPositionForDismissal),
    animated: false)
  DispatchQueue.main.asyncAfter(deadline: .now() + 0.2 , execute: {
    self.close()
  })
}
```

Build and run and pat yourself on the back for a job well done!

6) That's it!

Congrats, at this time you should understand how to create a snapshot image from a `UIView` and how to tap into scrolling properties to create smooth animations.