<https://www.rightpoint.com/rplabs/items-in-a-uicollectionview-animation>

**Animating Items in a UICollectionView**

[](https://www.rightpoint.com/rplabs/wp-content/uploads/sites/10/2015/01/coredata_rplaps-1024x346.png)

UICollectionView has been called “[the new UITableView](http://nshipster.com/uicollectionview/)“, which is not much of an exaggeration. The incredible flexibility of collection views for displaying and facilitating interaction with a collection of data items makes it one of the most valuable tools in the UIKit toolbox for iOS 6 and above. It provides a deeply customizable framework for displaying a collection of content views in practically any layout imaginable, as well as the ability to modify various display attributes of each view dynamically when scrolling through the collection, switching between distinct layouts, and inserting or removing views.

The ability to customize the animations that occur when inserting and removing items in a collection view – a very powerful tool for creating a dynamic user interface in an iOS app – has unfortunately not received much attention in existing literature and is not particularly well-documented. The process of customizing insertion/removal animations in a collection view is not particularly complex, but can be difficult to figure out from scratch.

Here are some of the key points that will be covered in this article:

* Both cells and supplementary views can be animated on insertion and removal.
* When inserting or removing a section explicitly, there is no need to also insert/remove individual items.
* In order to create the appropriate animations for a particular scenario, it is necessary to keep track of what happens in the layout invalidation process.
* Attributes can also be pre-computed or cached in an earlier part of the invalidation process.
* For the most fine-grained control over custom animations, it is best to create a fully custom layout.

Note: Given that there are many excellent resources available for learning collection view basics and creating full-fledged custom layouts, this article will not go into detail on those topics. Interested readers should refer to the [Resources](https://www.rightpoint.com/rplabs/items-in-a-uicollectionview-animation#Resources) section at the end of the article.

COLLECTION VIEW BASICS

**THE COLLECTION VIEW**

UICollectionView is a subclass of UIScrollView and, as such, it can support virtually infinite scrollable content. Each instance must have a valid data source (UICollectionViewDataSource), which is responsible for providing the section/item counts as well as the actual content for the collection view, and a layout (UICollectionViewLayout), which determines how content will appear on the screen. A collection view can also optionally be assigned a delegate (UICollectionViewDelegate), which can be implemented to prevent or respond to selection and deselection of cells, among other things.

**THE ITEMS**

There are several types of items which are managed by a collection view, all of which can be animated in some way or another.

Cells are instances of a subclass of [UICollectionViewCell](https://developer.apple.com/library/ios/documentation/uikit/reference/UICollectionViewCell_class/Reference/Reference.html). Analogous to table view cells, collection view cells are the primary view objects used to display the content in a collection view. Typically, each cell represents an individual data object which will be displayed to the user; for example, a photo with a caption.

Cells are created/dequeued, configured, and provided to the collection view by its data source, which must implement the following method:

Swift

|  |  |
| --- | --- |
| 1 | - (UICollectionViewCell \*)collectionView:(UICollectionView \*)collectionView cellForItemAtIndexPath:(NSIndexPath \*)indexPath |

Supplementary views are usually used to provide extra information about a section within the collection view, effectively replacing the concept of header and footer views in a table view.

Decoration views are used to provide visual ornamentation to the collection view (for example, a background image behind a section, or an ornamental icon at the end of the collection view).

Both supplementary and decoration views must be subclasses of [UICollectionReusableView](https://www.google.com/search?q=uicollectionreusableview&oq=uicollectionreus&aqs=chrome.1.69i57j0l3.5581j0&sourceid=chrome&ie=UTF-8).

All collection view item instances have a unique index path by which they can be identified in the collection view and its layout/delegate/data source. However, there is no correlation between the index paths of different item types, and items of different types may indeed have the same index path. According to the [documentation](https://developer.apple.com/library/ios/documentation/uikit/reference/UICollectionViewLayoutAttributes_class/Reference/Reference.html):

It is up to you to decide how to use the indexPath parameter to identify a given supplementary view. Typically, you use the elementKind parameter to identify the type of the supplementary view and the indexPath information to distinguish between different instances of that view.

**THE LAYOUT**

The layout characteristics of collection view items of any type are determined by the collection view’s layout object, which is an instance of a subclass of [UICollectionViewLayout](https://developer.apple.com/library/ios/documentation/uikit/reference/UICollectionViewLayout_class/Reference/Reference.html#//apple_ref/occ/instm/UICollectionViewLayout/layoutAttributesForDecorationViewOfKind:atIndexPath:). The layout determines which items are visible in a given bounding rectangle within the collection view’s scrollable content area and provides corresponding instances of [UICollectionViewLayoutAttributes](https://developer.apple.com/library/ios/documentation/uikit/reference/UICollectionViewLayoutAttributes_class/Reference/Reference.html) to determine each item’s positioning, transform, size, or any number of other custom attributes.

The default concrete layout subclass is [UICollectionViewFlowLayout](https://developer.apple.com/library/ios/documentation/uikit/reference/UICollectionViewFlowLayout_class/Reference/Reference.html), which evenly spaces rectangular items within the bounds of the collection view, depending on a number of provided constraints – vertical or horizontal scrolling, minimum item spacing, section insets, etc.

The layout is also where the attributes of items being inserted or removed can be customized to create dynamic and interesting animations.

INSERTING AND REMOVING ITEMS

UICollectionView provides several methods for explicitly animating position and content updates to cells in the collection, rather than simply calling reloadData and having it instantly change. This provides users of an application more visual feedback on how the collection is changing in response to new data becoming available or in response to an interaction with the content (deleting a photo, for example).

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | // Insert items    - (void)insertItemsAtIndexPaths:(NSArray \*)indexPaths;    // Insert sections    - (void)insertSections:(NSIndexSet \*)sections;    // Remove items    - (void)deleteItemsAtIndexPaths:(NSArray \*)indexPaths;    // Remove sections    - (void)deleteSections:(NSIndexSet \*)sections;    // Move item    - (void)moveItemAtIndexPath:(NSIndexPath \*)indexPath toIndexPath:(NSIndexPath \*)newIndexPath;    // Reload items    - (void)reloadItemsAtIndexPaths:(NSArray \*)indexPaths; |

**BATCH UPDATES**

UICollectionView provides a block-based method to synchronize cell animations and reduce the computational overhead involved with calculating and executing the animations for a group of updates:

Swift

|  |  |
| --- | --- |
| 1 | - (void)performBatchUpdates:(void (^)(void))updates completion:(void (^)(BOOL finished))completion; |

Any of the individual item insert/remove/move calls made from within the updates block will be precomputed and animated in-sync. After the animations stop, thecompletion block will be called, with the value of finished indicating whether the animations were interrupted or allowed to run to completion. This should feel familiar to anyone who has worked with UIView animations before.

**CAVEATS**

There are several important things to be aware of when performing batched updates on a collection view, some of which are not well-documented:

1. Much like UITableView, the collection view data source must be ready to provide the correct item/section counts to reflect any modifications being made prior to performing any animated updates.
2. When inserting a new section within the batch update block, one must not insert any items into that section – that will be handled implicitly. In fact, inserting items into a newly-inserted section in a batch update block creates “ghost” layers which get stuck in the collection view layer hierarchy. Presumably this is a just bug withUICollectionView, as this behavior is not documented and no exception is thrown.
3. When removing a section from within a batch update block, there is no need to remove the individual items first. Doing so does not result in any undesirable behavior; it is merely redundant.
4. Although not documented, reloadItemsAtIndexPaths: is rarely safe to call within a batch update block. Any other operations performed within the block that will result in the reloaded item(s) being displaced or animated in another way will cause an exception to be thrown for having multiple animations for the same item. This is most likely because “reloading” an item amounts to removing it and re-inserting it in-place, which will conflict with any other animations that affect the item.

**A NOTE ABOUT SUPPLEMENTARY VIEWS**

Supplementary views can also be animated upon insertion/removal, but only when the insertion/removal occurs as a result of the layout or data source changing. The layout is responsible for determining the number of supplementary views, by returning layout attributes for each visible supplementary view inlayoutAttributesForItemsInRect:, and the data source is responsible for creating and returning the supplementary view itself fromcollectionView:viewForSupplementaryElementOfKind:atIndexPath:.

Because of this, supplementary views cannot be dynamically inserted and removed from the collection view without explicitly invalidating the layout and/or reloading the data source. Any differences between the previous set of supplementary view layout attributes and the set of attributes after invalidation will be animated appropriately.

UICollectionViewFlowLayout has built-in supplementary view management for one header view and one footer view per section. The data source can return a supplementary header/footer view or nil fromcollectionView:viewForSupplementaryElementOfKind:atIndexPath: when appropriate to determine whether to show the header/footer or not. When performing an animated insert or removal of sections, any associated header/footer views are also animated.

CUSTOMIZING ANIMATIONS IN THE LAYOUT

The high-level display attributes of an item within a collection view are dictated by instances of UICollectionViewLayoutAttributes. When items are “appearing” or “disappearing”, the layout is given an opportunity to provide a different set of attributes for an item in its initial or final state of existence within the collection view. These initial or final attributes will be used along with the normal layout attributes for that particular item to create a fluid animation by “tweening” between the two sets of attributes.

The definition of “appearing” and “disappearing” within a collection view is not obvious, and can apply in any of the following cases.

**Appearing**

* Item inserted explicitly or via a section insert
* Item moved as a result of an insert at a lower index path
* Item “reloaded” as a result of an animated bounds change
* Whenever the heck the layout feels like it

**Disappearing**

* Item removed explicitly or via a section removal
* Item moved as a result of a removal at a lower index path
* Item “reloaded” as a result of an animated bounds change
* Whenever the heck the layout feels like it

Regarding the final bullet: the layout can interpret items as appearing or disappearing even if they don’t seem to be affected by a particular change to the layout. A reasonable explanation for this is that the layout is covering all its bases by making sure things are in the right place (by reloading item attributes) even if it ends up being unnecessary. Regardless, it is important to be prepared for this case when customizing a layout.

**METHODS TO OVERRIDE IN UICOLLECTIONVIEWLAYOUT**

In order to achieve custom attribute animations for appearing and disappearing items, one or more of the following methods should be overridden in a layout subclass. Bear in mind the scenarios outlined above for which an item may “appear” or “disappear”.

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | - (UICollectionViewLayoutAttributes\*)initialLayoutAttributesForAppearingItemAtIndexPath:(NSIndexPath \*)itemIndexPath;    - (UICollectionViewLayoutAttributes\*)finalLayoutAttributesForDisappearingItemAtIndexPath:(NSIndexPath \*)itemIndexPath;    - (UICollectionViewLayoutAttributes\*)initialLayoutAttributesForAppearingSupplementaryElementOfKind:(NSString \*)elementKind atIndexPath:(NSIndexPath \*)elementIndexPath;    - (UICollectionViewLayoutAttributes\*)finalLayoutAttributesForDisappearingSupplementaryElementOfKind:(NSString \*)elementKind atIndexPath:(NSIndexPath \*)elementIndexPath;    - (UICollectionViewLayoutAttributes \*)initialLayoutAttributesForAppearingDecorationElementOfKind:(NSString \*)elementKind atIndexPath:(NSIndexPath \*)elementIndexPath;    - (UICollectionViewLayoutAttributes \*)finalLayoutAttributesForDisappearingDecorationElementOfKind:(NSString \*)elementKind atIndexPath:(NSIndexPath \*)elementIndexPath; |

It is often the case that the particular initial or final layout attributes for the item will need to be dependent on *why* the item is appearing or disappearing, which is not possible to discern from the above methods alone. Thankfully,UICollectionViewLayout also provides two sets of methods that “sandwich” the layout’s requests for initial/final attributes. These methods can be used to precompute or cache any necessary information to determine the attributes for appearing or disappearing items for a particular change to the collection view.

**Layout update due to bounds change**

When the bounds of the layout change and an invalidation is triggered, the following methods are called.

Swift

|  |  |
| --- | --- |
| 1  2  3 | - (void)prepareForAnimatedBoundsChange:(CGRect)oldBounds    - (void)finalizeAnimatedBoundsChange |

**Layout update due to items changing**

When items or sections are explicitly inserted, removed, moved, or reloaded, these methods are called any initial/final layout attributes are requested.

Swift

|  |  |
| --- | --- |
| 1  2  3 | - (void)prepareForCollectionViewUpdates:(NSArray \*)updateItems;    - (void)finalizeCollectionViewUpdates; |

The updateItems array contains instances of [UICollectionViewUpdateItem](https://developer.apple.com/library/ios/documentation/uikit/reference/UICollectionViewUpdateItem_class/Reference/Reference.html#//apple_ref/occ/cl/UICollectionViewUpdateItem) objects which represent each change in the current batch of updates.

**Item vs. Section Update (IMPORTANT)**

An instance of UICollectionViewUpdateItem can represent either an item or a section update, and the only way to tell which is that update items which represent section updates contain index paths with a valid section index and NSNotFound for the item index, while update items that represent item updates contain index paths with valid item indexes.

This appears to be completely undocumented by Apple, but it is absolutely consistent behavior.

**STEP-BY-STEP**

Let’s consider the case where a cell at index path [0, 1] is removed from a collection view with three items.

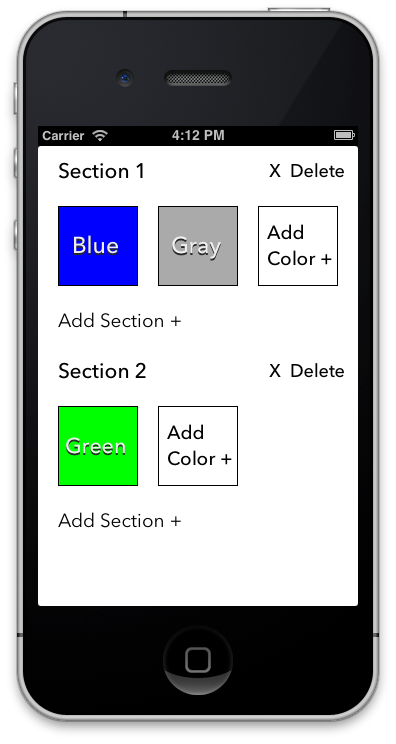
1. Update the data source so that it will return the new item at the correct index path and the correct number of items for the section.
2. Call deleteItemsAtIndexPaths: on the collection view with an index path equal to [0, 1] – second item in the first section.
3. The layout receives prepareLayout.
4. The layout receives prepareForCollectionViewUpdates: with an array containing one update item representing the deleted item.
5. The layout receives finalLayoutAttributesForDisappearingItemAtIndexPath: with index path [0, 1] – this is in reference to the deleted item being removed from the layout completely.
6. The layout receives finalLayoutAttributesForDisappearingItemAtIndexPath: with index path [0, 2] – this is in reference to the item previously at [0, 2] “disappearing” as it moves to [0, 1]
7. The layout receives initialLayoutAttributesForAppearingItemAtIndexPath: with index path [0, 1] – this is in reference to the item previously at [0, 2] “appearing” as it moves to [0, 1]
8. The layout receives finalizeCollectionViewUpdates.
9. The layout animates the cells to their new positions based on the attributes returned in steps 5-7.

This is the basic sequence of method calls received by a layout subclass when items are updated. As seen in the full example below, the real trick is to keep track of the state of the items in a collection view in order to return the correct attributes for any given scenario.

EXAMPLE PROJECT

An example project implementing custom item animations in a collection view can be downloaded [here](http://rz-cdn.raizlabs.com/dev/CollectionViewItemAnimations.zip).

For the sake of simplicity, this project uses a subclass ofUICollectionViewFlowLayout, which has many of its own behavioral quirks that must be taken into account to produce the desired effect when items are animated. Even still, there are some cases when the animations do not behave as expected due to the black box internals of UICollectionViewFlowLayout. A fully custom layout is recommended to achieve the most fine grained control over collection view item animations.



Tapping on “Add Color” will animate a new random color swatch into a section. Tapping on a color swatch will delete it, again with a custom animation. Tapping on “Add Section” will animate in a new section, and once there is more than one section, tapping “Delete” will (you guessed it) delete the section with yet another custom animation.

Let’s break down the code a bit.

**THE VIEW CONTROLLER**

ColorCollectionViewController is the main view controller class in the example project. It displays a collection view, implements bothUICollectionViewDataSource and UICollectionViewDelegateFlowLayout, and manages the user-initiated insertion and deletion of items and sections in the collection view.

The setup in viewDidLoad: is fairly straightforward. A collection view is created, initialized, and added to the view controller’s view:

Objective-C

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | // Allocate and configure the layout.  LessBoringFlowLayout \*layout = [[LessBoringFlowLayout alloc] init];  layout.minimumInteritemSpacing = 20.f;  layout.minimumLineSpacing = 20.f;  layout.scrollDirection = UICollectionViewScrollDirectionVertical;  layout.sectionInset = UIEdgeInsetsMake(10.f, 20.f, 10.f, 20.f);    // Bigger items for iPad  layout.itemSize = UI\_USER\_INTERFACE\_IDIOM() == UIUserInterfaceIdiomPad ? CGSizeMake(120, 120) : CGSizeMake(80, 80);    // uncomment this to see the default flow layout behavior  //[layout makeBoring];    // Allocate and configure a collection view  UICollectionView \* collectionView = [[UICollectionView alloc] initWithFrame:self.view.bounds     collectionViewLayout:layout];  collectionView.autoresizingMask = UIViewAutoresizingFlexibleWidth | UIViewAutoresizingFlexibleHeight;  collectionView.backgroundColor = [UIColor whiteColor];  collectionView.bounces = YES;  collectionView.alwaysBounceVertical = YES;  collectionView.dataSource = self;  collectionView.delegate = self;  self.collectionView = collectionView;    // Register reusable items  [collectionView registerNib:[UINib nibWithNibName:NSStringFromClass([AddCell class]) bundle:nil]  forCellWithReuseIdentifier:NSStringFromClass([AddCell class])];    [collectionView registerNib:[UINib nibWithNibName:NSStringFromClass([ColorNameCell class]) bundle:nil]  forCellWithReuseIdentifier:NSStringFromClass([ColorNameCell class])];    [collectionView registerNib:[UINib nibWithNibName:NSStringFromClass([ColorSectionHeaderView class]) bundle:nil]  forSupplementaryViewOfKind:UICollectionElementKindSectionHeader          withReuseIdentifier:NSStringFromClass([ColorSectionHeaderView class])];  [collectionView registerNib:[UINib nibWithNibName:NSStringFromClass([ColorSectionFooterView class]) bundle:nil]  forSupplementaryViewOfKind:UICollectionElementKindSectionFooter          withReuseIdentifier:NSStringFromClass([ColorSectionFooterView class])];    // Add to view  [self.view addSubview:collectionView]; |

Note that uncommenting [layout makeBoring] will revert the layout to a vanillaUICollectionViewFlowLayout for the purposes of comparing the custom animations to the default animations.

Another detail worth noting in the view controller is the forced data reload on the completion of the update block for adding or removing a section to the collection view, found in insertSectionAtIndex: and deleteSectionAtIndex::

Objective-C

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // Batch this so the other sections will be updated on completion  [self.collectionView performBatchUpdates:^{      [self.collectionView insertSections:[NSIndexSet indexSetWithIndex:index]];  }                                completion:^(BOOL finished) {                                    [self.collectionView reloadData];                                }]; |

Unlike cells, there is no way to retrieve a particular supplementary view from a collection view by its index path. Since supplementary views are recycled the same way as cells, it is also not a good idea to hold onto a reference to any particular supplementary view instance. Hence, there isn’t really a reliable way to update a single supplementary view dynamically. For this reason, it is necessary to callreloadData in order to update a supplementary view by letting the data source reconfigure it. In this case, the section headers are being reloaded to update the section number and the visibility of the “delete” button.

**THE LAYOUT**

For lack of a better name, the flow layout subclass in the example project is calledLessBoringFlowLayout. This layout is a direct subclass ofUICollectionViewFlowLayout with no modifications to the positioning or sizing of cells. The only difference comes in the form of customized insertion/removal animations for cells and supplementary views.

**Tracking Updates**

In order to distinguish between different scenarios in which the layout requests initial and final attributes for appearing and disappearing items, it is necessary to track when items and sections are added and removed from the collection view.

For this purpose, there are four mutable array properties in the class continuation interface that will be used to keep track of the index paths and section indexes of modified items and sections, respectively.

Objective-C

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Containers for keeping track of changing items  @property (nonatomic, strong) NSMutableArray \*insertedIndexPaths;  @property (nonatomic, strong) NSMutableArray \*removedIndexPaths;  @property (nonatomic, strong) NSMutableArray \*insertedSectionIndices;  @property (nonatomic, strong) NSMutableArray \*removedSectionIndices; |

These arrays are allocated and populated in prepareForCollectionViewUpdates:

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | - (void)prepareForCollectionViewUpdates:(NSArray \*)updateItems  {      [super prepareForCollectionViewUpdates:updateItems];        if (!self.isBoring)      {          // Keep track of updates to items and sections so we can use this information to create nifty animations          self.insertedIndexPaths     = [NSMutableArray array];          self.removedIndexPaths      = [NSMutableArray array];          self.insertedSectionIndices = [NSMutableArray array];          self.removedSectionIndices  = [NSMutableArray array];            [updateItems enumerateObjectsUsingBlock:^(UICollectionViewUpdateItem \*updateItem, NSUInteger idx, BOOL \*stop) {              if (updateItem.updateAction == UICollectionUpdateActionInsert)              {                  // If the update item's index path has an "item" value of NSNotFound, it means it was a section update, not an individual item.                  // This is 100% undocumented but 100% reproducible.                    if (updateItem.indexPathAfterUpdate.item == NSNotFound)                  {                      [self.insertedSectionIndices addObject:@(updateItem.indexPathAfterUpdate.section)];                  }                  else                  {                      [self.insertedIndexPaths addObject:updateItem.indexPathAfterUpdate];                  }              }              else if (updateItem.updateAction == UICollectionUpdateActionDelete)              {                  if (updateItem.indexPathBeforeUpdate.item == NSNotFound)                  {                      [self.removedSectionIndices addObject:@(updateItem.indexPathBeforeUpdate.section)];                    }                  else                  {                      [self.removedIndexPaths addObject:updateItem.indexPathBeforeUpdate];                  }              }          }];      }  } |

And they are deallocated in finalizeCollectionViewUpdates:

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | - (void)finalizeCollectionViewUpdates  {      [super finalizeCollectionViewUpdates];        self.insertedIndexPaths     = nil;      self.removedIndexPaths      = nil;      self.insertedSectionIndices = nil;      self.removedSectionIndices  = nil;  } |

**Caching Attributes**

Because UICollectionViewFlowLayout apparently performs its own internal caching, calls to super methods to obtain layout attributes for a particular index path will often return unexpected results. To bypass this behavior when creating a subclass with custom item animations, we can cache the current and previous set of item and supplementary view layout attributes as they are returned fromlayoutAttributesForElementsInRect:.

Cache Container Properties

The attributes for cells and supplementary views are kept separate. The cell layout attribute instances are hashed by cell index path. The supplementary view layout attribute instances are hashed first by element kind (an NSString), for which the value is another mutable dictionary, wherein the layout attribute instances are hashed by index path.

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Caches for keeping current/previous attributes  @property (nonatomic, strong) NSMutableDictionary \*currentCellAttributes;  @property (nonatomic, strong) NSMutableDictionary \*currentSupplementaryAttributesByKind;  @property (nonatomic, strong) NSMutableDictionary \*cachedCellAttributes;  @property (nonatomic, strong) NSMutableDictionary \*cachedSupplementaryAttributesByKind; |

Caching the Attributes from the Previous Invalidation

At the start of the layout invalidation, the layout receives a call to prepareLayout, in which the previous collection of layout attributes is cached via a deep copy to a separate set of containers. These attributes represent the previous location of each item by its previous index path.

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | - (void)prepareLayout  {      [super prepareLayout];        // Deep-copy attributes in current cache      self.cachedCellAttributes = [[NSMutableDictionary alloc] initWithDictionary:self.currentCellAttributes copyItems:YES];      self.cachedSupplementaryAttributesByKind = [NSMutableDictionary dictionary];      [self.currentSupplementaryAttributesByKind enumerateKeysAndObjectsUsingBlock:^(NSString \*kind, NSMutableDictionary \* attribByPath, BOOL \*stop) {          NSMutableDictionary \* cachedAttribByPath = [[NSMutableDictionary alloc] initWithDictionary:attribByPath copyItems:YES];          [self.cachedSupplementaryAttributesByKind setObject:cachedAttribByPath forKey:kind];      }];  } |

Updating the Current Attribute Cache

After receiving a call to prepareLayout, the layout will receive a call tolayoutAttributesForElementsInRect. The super implementation provided by the flow layout will handle computing the attributes for any cells and supplementary views in the collection view, which are added in-place to the mutable dictionary caches. Note that these caches are never emptied, merely overwritten and updated.

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | - (NSArray\*)layoutAttributesForElementsInRect:(CGRect)rect  {      NSArray \* attributes = [super layoutAttributesForElementsInRect:rect];        // Always cache all visible attributes so we can use them later when computing final/initial animated attributes      // Never clear the cache as certain items may be removed from the attributes array prior to being animated out      [attributes enumerateObjectsUsingBlock:^(UICollectionViewLayoutAttributes \*attributes, NSUInteger idx, BOOL \*stop) {            if (attributes.representedElementCategory == UICollectionElementCategoryCell)          {              [self.currentCellAttributes setObject:attributes                                             forKey:attributes.indexPath];          }          else if (attributes.representedElementCategory == UICollectionElementCategorySupplementaryView)          {              NSMutableDictionary \*supplementaryAttribuesByIndexPath = [self.currentSupplementaryAttributesByKind objectForKey:attributes.representedElementKind];              if (supplementaryAttribuesByIndexPath == nil)              {                  supplementaryAttribuesByIndexPath = [NSMutableDictionary dictionary];                  [self.currentSupplementaryAttributesByKind setObject:supplementaryAttribuesByIndexPath forKey:attributes.representedElementKind];              }                [supplementaryAttribuesByIndexPath setObject:attributes                                                    forKey:attributes.indexPath];          }        }];        return attributes;  } |

**Animation (Initial/Final) Layout Attributes**

Now that we know which items and sections are being updated as part of the current layout invalidation, as well as the current and previous “normal” layout attributes for each cell and supplementary view, we can implement the appropriateinitialLayoutAttributes… and finalLayoutAttributes… methods to create custom animations for inserted and removed items and sections.

Cells

For inserted and removed item index paths, corresponding to cells, the initial and final layout attribute methods are implemented as follows.

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | - (UICollectionViewLayoutAttributes\*)initialLayoutAttributesForAppearingItemAtIndexPath:(NSIndexPath \*)itemIndexPath  {      UICollectionViewLayoutAttributes \*attributes = [super initialLayoutAttributesForAppearingItemAtIndexPath:itemIndexPath];        if (!self.isBoring)      {          if ([self.insertedIndexPaths containsObject:itemIndexPath])          {              // If this is a newly inserted item, make it grow into place from its nominal index path              attributes = [[self.currentCellAttributes objectForKey:itemIndexPath] copy];              attributes.transform3D = CATransform3DMakeScale(0.1, 0.1, 1.0);          }          else if ([self.insertedSectionIndices containsObject:@(itemIndexPath.section)])          {              // if it's part of a new section, fly it in from the left              attributes = [[self.currentCellAttributes objectForKey:itemIndexPath] copy];              attributes.transform3D = CATransform3DMakeTranslation(-self.collectionView.bounds.size.width, 0, 0);          }        }      return attributes;  }    - (UICollectionViewLayoutAttributes\*)finalLayoutAttributesForDisappearingItemAtIndexPath:(NSIndexPath \*)itemIndexPath  {      UICollectionViewLayoutAttributes \*attributes = [super finalLayoutAttributesForDisappearingItemAtIndexPath:itemIndexPath];        if (!self.isBoring)      {            if ([self.removedIndexPaths containsObject:itemIndexPath] || [self.removedSectionIndices containsObject:@(itemIndexPath.section)])          {              // Make it fall off the screen with a slight rotation              attributes = [[self.cachedCellAttributes objectForKey:itemIndexPath] copy];              CATransform3D transform = CATransform3DMakeTranslation(0, self.collectionView.bounds.size.height, 0);              transform = CATransform3DRotate(transform, M\_PI\*0.2, 0, 0, 1);              attributes.transform3D = transform;              attributes.alpha = 0.0f;          }      }      return attributes;  } |

First, note that each method obtains the layout attributes object provided by the superclass. If not nil, these will represent the flow layout’s default animation attributes for the cell being inserted or removed.

For the initial layout attributes of an appearing cell, there are two cases:

1. The cell is being inserted into an existing section.
2. The cell has been inserted as a result of an entire section being inserted.

For case 1, we merely need to check if our stored array of index paths contains the index path passed in as an argument. If so, that means that the cell at that index path was explicitly inserted into a section. For this case, the animation will be implemented so the new item “grows” into place at its final location. To accomplish this, we extract the layout attributes object from our current attributes cache, copy it as not to mutate the cached instance, and apply a scale transform so that the item starts out very small, centered in its final location:

Swift

|  |  |
| --- | --- |
| 1  2 | attributes = [[self.currentCellAttributes objectForKey:itemIndexPath] copy];  attributes.transform3D = CATransform3DMakeScale(0.1, 0.1, 1.0); |

For case 2, we simply check if the passed-in index path’s section index matches an index in the array of inserted section indices. For this case, the animation will match the animation for the header/footer views in the section – all items will be animated into view from offscreen on the left, which is accomplished via a translation transform:

Swift

|  |  |
| --- | --- |
| 1  2 | attributes = [[self.currentCellAttributes objectForKey:itemIndexPath] copy];  attributes.transform3D = CATransform3DMakeTranslation(-self.collectionView.bounds.size.width, 0, 0); |

For the final layout attributes of a disappearing cell, the animation will be the same regardless of whether the cell was deleted individually or its entire section was deleted, so we can combine both cases together. All deleted items will fall off the bottom of the screen with a slight rotation and fade to clear, implemented with a transform and alpha change to the attributes object. Here, we need to use the previous cached attribues for the passed-in index path, since the current attributes cache will have a different attributes object for this index path.

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5 | attributes = [[self.cachedCellAttributes objectForKey:itemIndexPath] copy];  CATransform3D transform = CATransform3DMakeTranslation(0, self.collectionView.bounds.size.height, 0);  transform = CATransform3DRotate(transform, M\_PI\*0.2, 0, 0, 1);  attributes.transform3D = transform;  attributes.alpha = 0.0f; |

For either the initial or final attribute methods for cells, if the index path does not match an inserted or removed item or section, the superclass’s result is returned. Returning nil is also valid, although the effects of doing so seem to vary from case to case, at least for flow layout subclasses.

Supplementary Views (Header/Footer)

Calculating the header and footer views’ initial and final attributes is slightly more complicated. UICollectionViewFlowLayout will request initial and final layout attributes for supplementary views seemingly at random – including when only inserting/removing a cell – and returning the default flow layout’s initial/final attributes causes strange artifacts when a supplementary view had previously changed position. For this reason, the implementation in the example project is not perfect; a fully-custom layout is recommended for the highest degree of control over supplementary view animations.

The following two methods are implemented in the example project to create custom animations for header and footer supplementary views when the corresponding section is inserted or removed:

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | - (UICollectionViewLayoutAttributes\*)initialLayoutAttributesForAppearingSupplementaryElementOfKind:(NSString \*)elementKind atIndexPath:(NSIndexPath \*)elementIndexPath  {      UICollectionViewLayoutAttributes \*attributes = [super initialLayoutAttributesForAppearingSupplementaryElementOfKind:elementKind atIndexPath:elementIndexPath];        if (!self.isBoring)      {          if ([self.insertedSectionIndices containsObject:@(elementIndexPath.section)])          {              attributes = [[[self.currentSupplementaryAttributesByKind objectForKey:elementKind] objectForKey:elementIndexPath] copy];              attributes.transform3D = CATransform3DMakeTranslation(-self.collectionView.bounds.size.width, 0, 0);          }          else          {              // Start it right where it was before if nothing is changing              NSIndexPath \*prevPath = [self previousIndexPathForIndexPath:elementIndexPath accountForItems:NO];              attributes = [[[self.cachedSupplementaryAttributesByKind objectForKey:elementKind] objectForKey:prevPath] copy];          }        }        return attributes;  }    - (UICollectionViewLayoutAttributes\*)finalLayoutAttributesForDisappearingSupplementaryElementOfKind:(NSString \*)elementKind atIndexPath:(NSIndexPath \*)elementIndexPath  {      UICollectionViewLayoutAttributes \*attributes = [super finalLayoutAttributesForDisappearingSupplementaryElementOfKind:elementKind atIndexPath:elementIndexPath];        if (!self.isBoring)      {          if ([self.removedSectionIndices containsObject:@(elementIndexPath.section)])          {              attributes = [[[self.cachedSupplementaryAttributesByKind objectForKey:elementKind] objectForKey:elementIndexPath] copy];              CATransform3D transform = CATransform3DMakeTranslation(0, self.collectionView.bounds.size.height, 0);              transform = CATransform3DRotate(transform, M\_PI\*0.2, 0, 0, 1);              attributes.transform3D = transform;              attributes.alpha = 0.0f;          }          else          {              // Keep it right where it is              attributes = [[[self.currentCellAttributes objectForKey:elementKind] objectForKey:elementIndexPath] copy];          }        }      return attributes;  } |

For the initial layout attributes of a supplementary view, there are two cases to account for:

1. The section in which the supplementary resides was inserted.
2. All other cases – need to prevent the supplementary from moving around unnecessarily.

Case 1 is fairly straightforward – if the section in which the supplementary view resides was inserted, it should be animated appropriately (flown in to the right from off-screen):

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5 | if ([self.insertedSectionIndices containsObject:@(elementIndexPath.section)])  {      attributes = [[[self.currentSupplementaryAttributesByKind objectForKey:elementKind] objectForKey:elementIndexPath] copy];      attributes.transform3D = CATransform3DMakeTranslation(-self.collectionView.bounds.size.width, 0, 0);  } |

Case 2 is to attempt to counteract a strange side effect of creating a subclass ofUICollectionViewFlowLayout. After the size of a section changes due to an item insertion or deletion, the supplementary views will normally change position as a result. If the flow layout’s default initial attributes for supplementary views are returned for this animation, occasionally the view will appear to move from its previous position to its new position even if the size of the section remains constant. By returning the previously cached attributes for the same index path, the header/footer will stay in place:

Swift

|  |  |
| --- | --- |
| 1  2 | NSIndexPath \*prevPath = [self previousIndexPathForIndexPath:elementIndexPath accountForItems:NO];  attributes = [[[self.cachedSupplementaryAttributesByKind objectForKey:elementKind] objectForKey:prevPath] copy]; |

Here, a helper method is used to calculate the previous index path of an item in the collection view when sections or items are inserted or removed.

The other possible solutions to this issue are to use a fully-custom layout or not to animate supplementary views at all. In other scenarios, such as a section moving due to an earlier section being removed or inserted, the attributes returned from the default flow layout implementation are adequate to move the supplementary views in an animated fashion.

For the final layout attributes of a supplementary view, the cases are the same. However, the animation for a supplementary view being removed will be the same as a cell being removed:

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | if ([self.removedSectionIndices containsObject:@(elementIndexPath.section)])  {      attributes = [[[self.cachedSupplementaryAttributesByKind objectForKey:elementKind] objectForKey:elementIndexPath] copy];      CATransform3D transform = CATransform3DMakeTranslation(0, self.collectionView.bounds.size.height, 0);      transform = CATransform3DRotate(transform, M\_PI\*0.2, 0, 0, 1);      attributes.transform3D = transform;      attributes.alpha = 0.0f;  } |

And the current index path and cached attributes of the supplementary view can be used to keep it in place when necessary, rather than the previous ones:

Swift

|  |  |
| --- | --- |
| 1 | attributes = [[[self.currentCellAttributes objectForKey:elementKind] objectForKey:elementIndexPath] copy]; |

SUMMARY

UICollectionView is an incredibly powerful and flexible UI component for iOS applications. Customizing its behavior is not always straightforward, but is often well worth the effort. In particular, custom animations when inserting or removing items and sections from a collection view can be used to provide visual feedback to a user in order to better indicate the effects of an action within an app.

Thanks for reading!