Sample Code

Browsing and Modifying Photo Albums

Help users organize their photos into albums and browse photo collections in a grid-based layout using PhotoKit.

Download

iOS 11.0+ | iPadOS 11.0+ | tvOS 13.2+ | Xcode 11.3+

Overview

The Photos app on iOS displays assets in a thumbnail grid. This sample demonstrates how to achieve a similar layout with a custom `UICollectionViewController`. It fetches asset thumbnails using PhotoKit, then displays them as a single photo, video, or Live Photo asset.

The sample app, PhotoBrowse, also demonstrates how to organize the user's photos into albums and built-in collections, such as Recently Added and Favorites. It supports album creation, deletion, modification, as well as the editing and favoriting of individual assets.

Getting Started

The sample app, PhotoBrowse, runs on iOS and tvOS and requires the following:

- iOS 12 or later.
- tvOS 12 or later.

If you build and run the project in Xcode, you may also add photos to the app in Simulator.

List Albums and Built-in Collections

When the app first launches, it fetches all of the user's photo assets. All requests for asset data—single photos, albums, and user collections—go through PhotoKit's shared `PHPhotoLibrary`

object, so the app registers the main view controller once its view loads:

```
PHPhotoLibrary.shared().register(self)
```

To list all the user's albums and collections, the app creates a `PHFetchOptions` object and dispatches several fetch requests:

```
let allPhotosOptions = PHFetchOptions()
allPhotosOptions.sortDescriptors = [NSSortDescriptor(key: "creationDate", ascending:
allPhotos = PHAsset.fetchAssets(with: allPhotosOptions)
smartAlbums = PHAssetCollection.fetchAssetCollections(with: .smartAlbum, subtype: .a
userCollections = PHCollectionList.fetchTopLevelUserCollections(with: nil)
```

The resulting `PHFetchResult` informs the app about the structure of the user's photo library, allowing the user interface to show the number of photos in each album.

Display Assets in a Thumbnail Grid

The sample app implements thumbnail grid browsing by subclassing `UICollectionViewController` in `AssetGridViewController`.

After the grid view controller loads, it updates the image cache, which allows thumbnails to load quickly as the user scrolls:

```
let (addedRects, removedRects) = differencesBetweenRects(previousPreheatRect, prehea
let addedAssets = addedRects
    .flatMap { rect in collectionView!.indexPathsForElements(in: rect) }
    .map { indexPath in fetchResult.object(at: indexPath.item) }
let removedAssets = removedRects
    .flatMap { rect in collectionView!.indexPathsForElements(in: rect) }
    .map { indexPath in fetchResult.object(at: indexPath.item) }

// Update the assets the PHCachingImageManager is caching.
imageManager.startCachingImages(for: addedAssets,
                                targetSize: thumbnailSize, contentMode: .aspectFill,
imageManager.stopCachingImages(for: removedAssets,
                               targetSize: thumbnailSize, contentMode: .aspectFill,
```

Implement the `UICollectionView` delegate method `cellForItem(at:)` to use thumbnails instead of full assets. PhotoKit allows you to request assets directly, and even badge Live Photos

to set them apart:

```
// Dequeue a GridViewCell.
guard let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "GridViewCe
    else { fatalError("Unexpected cell in collection view") }

// Add a badge to the cell if the PHAsset represents a Live Photo.
if asset.mediaSubtypes.contains(.photoLive) {
    cell.livePhotoBadgeImage = PHLivePhotoView.livePhotoBadgeImage(options: .overCor
}

// Request an image for the asset from the PHCachingImageManager.
cell.representedAssetIdentifier = asset.localIdentifier
imageManager.requestImage(for: asset, targetSize: thumbnailSize, contentMode: .aspec
    // UIKit may have recycled this cell by the handler's activation time.
    // Set the cell's thumbnail image only if it's still showing the same asset.
    if cell.representedAssetIdentifier == asset.localIdentifier {
        cell.thumbnailImage = image
    }
})
```

Show a Single Photo, Video, or Live Photo

`AssetViewController` implements the view of a single asset. If the asset is a video or Live Photo, the view controller also supports playback through a `UIBarButtonItem`:

```
// Set the appropriate toolbar items based on the media type of the asset.
#if os(iOS)
navigationController?.isToolbarHidden = false
navigationController?.hidesBarsOnTap = true
if asset.mediaType == .video {
    toolbarItems = [favoriteButton, space, playButton, space, trashButton]
} else {
    // In iOS, present both stills and Live Photos the same way, because
    // PHLivePhotoView provides the same gesture-based UI as in the Photos app.
    toolbarItems = [favoriteButton, space, trashButton]
}
#elseif os(tvOS)
if asset.mediaType == .video {
    navigationItem.leftBarButtonItems = [playButton, favoriteButton, trashButton]
} else {
```

```
    // In tvOS, PHLivePhotoView doesn't support playback gestures,
    // so add a play button for Live Photos.
    if asset.mediaSubtypes.contains(.photoLive) {
        navigationItem.leftBarButtonItems = [favoriteButton, trashButton]
    } else {
        navigationItem.leftBarButtonItems = [livePhotoPlayButton, favoriteButton, tr
    }
}
#endif
```

On tvOS, PhotoKit supports Live Photo playback:

```
#if os(tvOS)
@IBAction func playLivePhoto(_ sender: Any) {
    livePhotoView.startPlayback(with: .full)
}
#endif
```

The view controller supports playback by creating an `AVPlayer` and layering it on top of the item once the `PHImageManager` fetches a video:

```
// Request an AVPlayerItem for the displayed PHAsset.
// Then configure a layer for playing it.
PHImageManager.default().requestPlayerItem(forVideo: asset, options: options, result
    DispatchQueue.main.sync {
        guard self.playerLayer == nil else { return }

        // Create an AVPlayer and AVPlayerLayer with the AVPlayerItem.
        let player = AVPlayer(playerItem: playerItem)
        let playerLayer = AVPlayerLayer(player: player)

        // Configure the AVPlayerLayer and add it to the view.
        playerLayer.videoGravity = AVLayerVideoGravity.resizeAspect
        playerLayer.frame = self.view.layer.bounds
        self.view.layer.addSublayer(playerLayer)

        player.play()

        // Cache the player layer by reference, so you can remove it later.
        self.playerLayer = playerLayer
    }
})
```

Apply Canned Filters in an Editing Interface

The `AssetViewController` view allows the user to edit the photo and save changes back to the photo library. It uses an alert controller to display a list of preset editing options, including sepia tone, chrome, and revert.

```
// Allow editing only if the PHAsset supports edit operations.
if asset.canPerform(.content) {
    // Add actions for some canned filters.
    alertController.addAction(UIAlertAction(title: NSLocalizedString("Sepia Tone", c
                                            style: .default, handler: getFilter("CIS
    alertController.addAction(UIAlertAction(title: NSLocalizedString("Chrome", comme
                                            style: .default, handler: getFilter("CIP

    // Add actions to revert any edits that have been made to the PHAsset.
    alertController.addAction(UIAlertAction(title: NSLocalizedString("Revert", comme
                                            style: .default, handler: revertAsset))
}
// Present the UIAlertController.
present(alertController, animated: true)
```

Each option creates an edit request and prepares a `PHContentEditingOutput` to encapsulate edit results as data. A `PHAssetChangeRequest` object communicates the edit back to the user's photo library.

```
DispatchQueue.global(qos: .userInitiated).async {

    // Create adjustment data describing the edit.
    let adjustmentData = PHAdjustmentData(formatIdentifier: self.formatIdentifier,
                                          formatVersion: self.formatVersion,
                                          data: filterName.data(using: .utf8)!)

    // Create content editing output, write the adjustment data.
    let output = PHContentEditingOutput(contentEditingInput: input)
    output.adjustmentData = adjustmentData

    // Select a filtering function for the asset's media type.
    let applyFunc: (String, PHContentEditingInput, PHContentEditingOutput, @escaping
    if self.asset.mediaSubtypes.contains(.photoLive) {
        applyFunc = self.applyLivePhotoFilter
    } else if self.asset.mediaType == .image {
        applyFunc = self.applyPhotoFilter
```

```
    } else {
        applyFunc = self.applyVideoFilter
    }

    // Apply the filter.
    applyFunc(filterName, input, output, {
        // When the app finishes rendering the filtered result, commit the edit to t
        PHPhotoLibrary.shared().performChanges({
            let request = PHAssetChangeRequest(for: self.asset)
            request.contentEditingOutput = output
        }, completionHandler: { success, error in
            if !success { print("Can't edit the asset: \(String(describing: error))'
        })
    })
}
```

After the user picks a filter, the app applies it and outputs the saved asset immediately. There's no UI state for having chosen—but not yet committed—an edit. As such, there's no role for reading adjustment data to resume in-progress edits, since PhotoBrowse has no notion of *in-progress*. However, it's still good practice to write adjustment data so that potential future versions of the app—or other apps that understand your adjustment data format—could make use of it.

Create a New Album

An alert controller allows the user to add a new album:

```
PHPhotoLibrary.shared().performChanges({
    PHAssetCollectionChangeRequest.creationRequestForAssetCollection(withTitle: titl
}, completionHandler: { success, error in
    if !success { print("Error creating album: \(String(describing: error)).") }
})
```

Add an Asset to a Collection

When the user chooses to add an asset by tapping the Add button (+) in the navigation bar, PhotoBrowse creates a mock photo from a random color at a random orientation. Like other changes to a user's photo library, adding an asset requires the app to wrap the addition inside a `PHAssetChangeRequest` as follows:

```
// Add the asset to the photo library.
PHPhotoLibrary.shared().performChanges({
    let creationRequest = PHAssetChangeRequest.creationRequestForAsset(from: image)
    if let assetCollection = self.assetCollection {
        let addAssetRequest = PHAssetCollectionChangeRequest(for: assetCollection)
        addAssetRequest?.addAssets([creationRequest.placeholderForCreatedAsset!] as
    }
}, completionHandler: {success, error in
    if !success { print("Error creating the asset: \(String(describing: error))") }
})
```

Delete Assets and Albums

The user can delete an asset through the trash can button at the lower-right corner of `AssetViewController`. For removal from an album, PhotoBrowse wraps the deletion operation inside a `PHAssetCollectionChangeRequest` object. For removal from the entire photo library, PhotoBrowse wraps the deletion operation inside a `PHAssetChangeRequest` object:

```
if assetCollection != nil {
    // Remove the asset from the selected album.
    PHPhotoLibrary.shared().performChanges({
        let request = PHAssetCollectionChangeRequest(for: self.assetCollection)!
        request.removeAssets([self.asset as Any] as NSArray)
    }, completionHandler: completion)
} else {
    // Delete the asset from the photo library.
    PHPhotoLibrary.shared().performChanges({
        PHAssetChangeRequest.deleteAssets([self.asset as Any] as NSArray)
    }, completionHandler: completion)
}
```

Favorite an Asset

Users can favorite an asset by toggling the `PHAsset` parameter `isFavorite`:

```
PHPhotoLibrary.shared().performChanges({
    let request = PHAssetChangeRequest(for: self.asset)
    request.isFavorite = !self.asset.isFavorite
}, completionHandler: { success, error in
    if success {
```

```
            DispatchQueue.main.sync {
                sender.title = self.asset.isFavorite ? "♥" : "♡"
            }
        } else {
            print("Can't mark the asset as a Favorite: \(String(describing: error))")
        }
    })
```

Observe and Respond to Changes

Register your main view controller—and any view that shows the user assets—to observe changes to the photo library, so your app can receive and respond to notifications as assets change. These changes may not necessarily occur inside your app's functionality; they could originate from other apps, other devices, iCloud Photos, or Shared Albums:

```
if let changeDetails = changeInstance.changeDetails(for: allPhotos) {
    // Update the cached fetch result.
    allPhotos = changeDetails.fetchResultAfterChanges
    // Don't update the table row that always reads "All Photos."
}

// Update the cached fetch results, and reload the table sections to match.
if let changeDetails = changeInstance.changeDetails(for: smartAlbums) {
    smartAlbums = changeDetails.fetchResultAfterChanges
    tableView.reloadSections(IndexSet(integer: Section.smartAlbums.rawValue), with:
}
if let changeDetails = changeInstance.changeDetails(for: userCollections) {
    userCollections = changeDetails.fetchResultAfterChanges
    tableView.reloadSections(IndexSet(integer: Section.userCollections.rawValue), wi
}
```

Be sure to unregister your change observer after the app's main view controller goes away.

```
PHPhotoLibrary.shared().unregisterChangeObserver(self)
```

See Also

Sample code

{} Selecting Photos and Videos in iOS

Improve the user experience of finding and selecting assets by using the Photos picker.

{} Bringing Photos picker to your SwiftUI app

Select media assets by using a Photos picker view that SwiftUI provides.

{} Implementing an inline Photos picker

Embed a system-provided, half-height Photos picker into your app's view.

{} Creating a Slideshow Project Extension for Photos

Augment the macOS Photos app with extensions that support project creation.