Building iOS 17 Apps with Xcode Storyboards

Neil Smyth

Payload publishing

Building iOS 17 Apps with Xcode Storyboards

Building iOS 17 Apps with Xcode Storyboards

ISBN-13: 978-1-951442-84-2

© 2024 Neil Smyth / Payload Media, Inc. All Rights Reserved.

This book is provided for personal use only. Unauthorized use, reproduction and/or distribution strictly prohibited. All rights reserved.

The content of this book is provided for informational purposes only. Neither the publisher nor the author offers any warranties or representation, express or implied, with regard to the accuracy of information contained in this book, nor do they accept any liability for any loss or damage arising from any errors or omissions.

This book contains trademarked terms that are used solely for editorial purposes and to the benefit of the respective trademark owner. The terms used within this book are not intended as infringement of any trademarks.

Rev: 1.0

Payload
publishing

https://www.payloadbooks.com

65. iOS 17 UIKit Dynamics – An Overview

UIKit Dynamics provides a powerful and flexible mechanism for combining user interaction and animation into iOS user interfaces. What distinguishes UIKit Dynamics from other approaches to animation is the ability to declare animation behavior in terms of real-world physics.

Before moving on to a detailed tutorial in the next chapter, this chapter will provide an overview of the concepts and methodology behind UIKit Dynamics in iOS.

65.1 Understanding UIKit Dynamics

UIKit Dynamics allows for the animation of user interface elements (typically view items) to be implemented within a user interface, often in response to user interaction. To fully understand the concepts behind UIKit Dynamics, it helps to visualize how real-world objects behave.

Holding an object in the air and then releasing it, for example, will cause it to fall to the ground. This behavior is, of course, the result of gravity. However, whether or not, and by how much, an object bounces upon impact with a solid surface is dependent upon that object's elasticity and its velocity at the point of impact.

Similarly, pushing an object positioned on a flat surface will cause that object to travel a certain distance depending on the magnitude and angle of the pushing force combined with the level of friction at the point of contact between the two surfaces.

An object tethered to a moving point will react in various ways, such as following the anchor point, swinging in a pendulum motion, or even bouncing and spinning on the tether in response to more aggressive motions. However, an object similarly attached using a spring will behave entirely differently in response to the movement of the point of attachment.

Considering how objects behave in the real world, imagine the ability to selectively apply these same physics-related behaviors to view objects in a user interface, and you will begin understanding the basic concepts behind UIKit Dynamics. Not only does UIKit Dynamics allow user interface interaction and animation to be declared using concepts we are already familiar with, but in most cases, it allows this to be achieved with just a few simple lines of code.

65.2 The UIKit Dynamics Architecture

Before looking at how UIKit Dynamics are implemented in app code, it helps to understand the different elements that comprise the dynamics architecture.

The UIKit Dynamics implementation comprises four key elements: a *dynamic animator*, a set of one or more *dynamic behaviors*, one or more *dynamic items,* and a *reference view.*

65.2.1 Dynamic Items

The dynamic items are the view elements within the user interface to be animated in response to specified dynamic behaviors. A dynamic item is any view object that implements the *UIDynamicItem* protocol, which includes the UIView and UICollectionView classes and any subclasses thereof (such as UIButton and UILabel).

Any custom view item can work with UIKit Dynamics by conforming to the UIDynamicItem protocol.

65.2.2 Dynamic Behaviors

Dynamic behaviors are used to configure the behavior to be applied to one or more dynamic items. A range of predefined dynamic behavior classes is available, including *UIAttachmentBehavior*, *UICollisionBehavior*, *UIGravityBehavior*, *UIDynamicItemBehavior*, *UIPushBehavior,* and *UISnapBehavior.* Each is a subclass of the *UIDynamicBehavior* class, which will be covered in detail later in this chapter.

In general, an instance of the class corresponding to the desired behavior (UIGravityBehavior for gravity, for example) will be created, and the dynamic items for which the behavior is to be applied will be added to that instance. Dynamic items can be assigned to multiple dynamic behavior instances simultaneously and may be added to or removed from a dynamic behavior instance during runtime.

Once created and configured, behavior objects are added to the *dynamic animator* instance. Once added to a dynamic animator, the behavior may be removed at any time.

65.2.3 The Reference View

The reference view dictates the area of the screen within which the UIKit Dynamics animation and interaction are to take place. This is typically the parent superclass view or collection view, of which the dynamic item views are children.

65.2.4 The Dynamic Animator

The dynamic animator coordinates the dynamic behaviors and items and works with the underlying physics engine to perform the animation. The dynamic animator is represented by an instance of the *UIDynamicAnimator* class and is initialized with the corresponding reference view at creation time. Once created, suitably configured dynamic behavior instances can be added and removed as required to implement the desired user interface behavior.

The overall architecture for a UIKit Dynamics implementation can be represented visually using the diagram outlined in Figure 65-1:

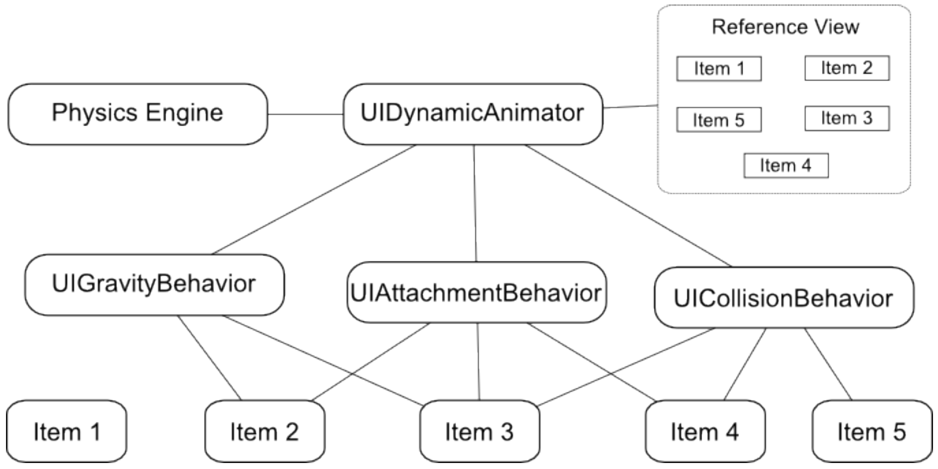

Figure 65-1

The above example has added three dynamic behaviors to the dynamic animator instance. The reference view contains five dynamic items, all but one of which have been added to at least one dynamic behavior instance.

65.3 Implementing UIKit Dynamics in an iOS App

The implementation of UIKit Dynamics in an app requires three very simple steps:

1. Create an instance of the *UIDynamicAnimator* class to act as the dynamic animator and initialize it with reference to the reference view.

2. Create and configure a dynamic behavior instance and assign to it the dynamic items on which the specified behavior is to be imposed.

3. Add the dynamic behavior instance to the dynamic animator.

4. Repeat from step 2 to create and add additional behaviors.

65.4 Dynamic Animator Initialization

The first step in implementing UIKit Dynamics is to create and initialize an instance of the UIDynamicAnimator class. The first step is to declare an instance variable for the reference:

```
var animator: UIDynamicAnimator?
```

Next, the dynamic animator instance can be created. The following code, for example, creates and initializes the animator instance within the *viewDidLoad* method of a view controller, using the view controller's parent view as the reference view:

```
override func viewDidLoad() {
    super.viewDidLoad()
    animator = UIDynamicAnimator(referenceView: self.view)
}
```

With the dynamic animator created and initialized, the next step is to configure behaviors, the details for which differ slightly depending on the nature of the behavior.

65.5 Configuring Gravity Behavior

Gravity behavior is implemented using the UIGravityBehavior class, the purpose of which is to cause view items to want to "fall" within the reference view as though influenced by gravity. UIKit Dynamics gravity is slightly different from real-world gravity in that it is possible to define a vector for the direction of the gravitational force using x and y components (x, y) contained within a CGVector instance. The default vector for this class is (0.0, 1.0), corresponding to downward acceleration at a speed of 1000 points per second^2. A negative x or y value will reverse the direction of gravity.

A UIGravityBehavior instance can be initialized as follows, passing through an array of dynamic items on which the behavior is to be imposed (in this case, two views named view1 and view2):

```
let gravity = UIGravityBehavior(items: [view1, view2])
```

Once created, the default vector can be changed if required at any time:

```
let vector = CGVectorMake(0.0, 0.5)
gravity.gravityDirection = vector
```

Finally, the behavior needs to be added to the dynamic animator instance:

```
animator?.addBehavior(gravity)
```

At any point during the app lifecycle, dynamic items may be added to, or removed from, the behavior:

```
gravity.addItem(view3)
gravity.removeItem(view)
```

Similarly, the entire behavior may be removed from the dynamic animator:

```
animator?.removeBehavior(gravity)
```

When gravity behavior is applied to a view, and in the absence of opposing behaviors, the view will immediately move in the direction of the specified gravity vector. In fact, as currently defined, the view will fall out of the bounds of the reference view and disappear. This can be prevented by setting up a collision behavior.

65.6 Configuring Collision Behavior

UIKit Dynamics is all about making items move on the device display. When an item moves, there is a high chance it will collide either with another item or the boundaries of the encapsulating reference view. As previously discussed, in the absence of any form of collision behavior, a moving item can move out of the visible area of the reference view. Such a configuration will also cause a moving item to simply pass over the top of any other items that happen to be in its path. Collision behavior (defined using the UICollisionBehavior class) allows such collisions to behave in ways more representative of the real world.

Collision behavior can be implemented between dynamic items (such that certain items can collide with others) or within boundaries (allowing collisions to occur when an item reaches a designated boundary). Boundaries can be defined such that they correspond to the boundaries of the reference view, or entirely new boundaries can be defined using lines and Bezier paths.

As with gravity behavior, a collision is generally created and initialized with an array object containing the items to which the behavior is to be applied. For example:

```
let collision = UICollisionBehavior(items: [view1, view2])
animator?.addBehavior(collision)
```

As configured, view1 and view2 will now collide when coming into contact. The physics engine will decide what happens depending on the items' elasticity and the collision's angle and speed. In other words, the engine will animate the items to behave as if they were physical objects subject to the laws of physics.

By default, an item under the influence of a collision behavior will collide with other items in the same collision behavior set and any boundaries set up. To declare the reference view as a boundary, set the *translatesReferenceBoundsIntoBoundary* property of the behavior instance to *true*:

```
collision.translatesReferenceBoundsIntoBoundary = true
```

A boundary inset from the edges of the reference view may be defined using the *setsTranslateReferenceBoundsIntoBoundaryWithInsets* method, passing through the required insets as an argument in the form of a *UIEdgeInsets* object.

The *collisionMode* property may be used to change default collision behavior by assigning one of the following constants:

- **UICollisionBehaviorMode.items** – Specifies that collisions only occur between items added to the collision behavior instance. Boundary collisions are ignored.
- **UICollisionBehaviorMode.boundaries** – Configures the behavior to ignore item collisions, recognizing only collisions with boundaries.
- **UICollisionBehaviorMode.everything** – Specifies that collisions occur between items added to the behavior and all boundaries. This is the default behavior.

The following code, for example, enables collisions only for items:

```
collision.collisionMode = UICollisionBehaviorMode.items
```

If an app needs to react to a collision, declare a class instance that conforms to the UICollisionBehaviorDelegate class by implementing the following methods and assign it as the delegate for the UICollisionBehavior object instance.

- collisionBehavior(_:beganContactForItem:withBoundaryIdentifier:atPoint:)
- collisionBehavior(_:beganContactForItem:withItem:atPoint:)
- collisionBehavior(_:endedContactForItem:withBoundaryIdentifier:)
- collisionBehavior(_:endedContactForItem:withItem:)

When implemented, the app will be notified when collisions begin and end. In most cases, the delegate methods will be passed information about the collision, such as the location and the items or boundaries involved.

In addition, aspects of the collision behavior, such as friction and the elasticity of the colliding items (such that they bounce on contact), may be configured using the UIDynamicBehavior class. This class will be covered in detail later in this chapter.

65.7 Configuring Attachment Behavior

As the name suggests, the UIAttachmentBehavior class allows dynamic items to be configured to behave as if attached. These attachments can take the form of two items attached or an item attached to an anchor point at specific coordinates within the reference view. In addition, the attachment can take the form of an imaginary piece of cord that does not stretch or a spring attachment with configurable damping and frequency properties that control how "bouncy" the attached item is in response to motion.

By default, the attachment point within the item itself is positioned at the center of the view. This can, however, be changed to a different position causing the real-world behavior outlined in Figure 65-2 to occur:

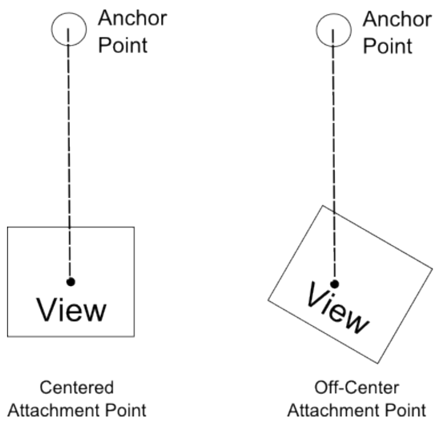

Figure 65-2

The physics engine will generally simulate animation to match what would typically happen in the real world. As illustrated above, the item will tilt when not attached in the center. If the anchor point moves, the attached view will also move. Depending on the motion, the item will swing in a pendulum motion and, assuming appropriate collision behavior configuration, bounce off any boundaries it collides with as it swings.

As with all UIKit Dynamics behavior, the physics engine performs all the work to achieve this. The only effort required by the developer is to write a few lines of code to set up the behavior before adding it to the dynamic animator instance. The following code, for example, sets up an attachment between two dynamic items:

```
let attachment = UIAttachmentBehavior(item: view1,
                                       attachedToItem: view2)
animator?.addBehavior(attachment)
```

The following code, on the other hand, specifies an attachment between view1 and an anchor point with the frequency and damping values set to configure a spring effect:

```
let anchorpoint = CGPoint(x: 100, y: 100)
let attachment = UIAttachmentBehavior(item: view1,
                    attachedToAnchor: anchorPoint)
attachment.frequency = 4.0
attachment.damping = 0.0
```

The above examples attach to the center point of the view. The following code fragment sets the same attachment as above, but with an attachment point offset 20, 20 points relative to the center of the view:

```
let anchorpoint = CGPoint(x: 100, y: 100)
let offset = UIOffset(horizontal: 20, vertical: 20)

let attachment = UIAttachmentBehavior(item: view1,
                          offsetFromCenter: offset,
                          attachedToAnchor: anchorPoint)
```

65.8 Configuring Snap Behavior

The UISnapBehavior class allows a dynamic item to be "snapped" to a specific location within the reference view. When implemented, the item will move toward the snap location as though pulled by a spring and, depending on the damping property specified, oscillate several times before finally snapping into place. Until the behavior is removed from the dynamic animator, the item will continue to snap to the location when subsequently moved to another position.

The damping property can be set to any value between 0.0 and 1.0, with 1.0 specifying maximum oscillation. The default value for damping is 0.5.

The following code configures snap behavior for dynamic item view1 with damping set to 1.0:

```
let point = CGPoint(x: 100, y: 100)
let snap = UISnapBehavior(item: view1, snapToPoint: point)
snap.damping = 1.0

animator?.addBehavior(snap)
```

65.9 Configuring Push Behavior

Push behavior, defined using the UIPushBehavior class, simulates the effect of pushing one or more dynamic items in a specific direction with a specified force. The force can be specified as continuous or instantaneous. In the case of a continuous push, the force is continually applied, causing the item to accelerate over time. The instantaneous push is more like a "shove" than a push in that the force is applied for a short pulse causing the item to gain velocity quickly but gradually lose momentum and eventually stop. Once an instantaneous push event has been completed, the behavior is disabled (though it can be re-enabled).

The direction of the push can be defined in radians or using x and y components. By default, the pushing force is applied to the center of the dynamic item, though, as with attachments, this can be changed to an offset relative to the center of the view.

A force of magnitude 1.0 is defined as being a force of one UIKit Newton, which equates to a view sized at 100 x 100 points with a density of value 1.0 accelerating at a rate of 100 points per second^2. As explained in the next section, the density of a view can be configured using the UIDynamicItemBehavior class.

The following code pushes an item with instantaneous force at a magnitude of 0.2 applied on both the x and y axes, causing the view to move diagonally down and to the right:

```
let push = UIPushBehavior(items: [view1],
                       mode: UIPushBehaviorMode.instantaneous)
let vector = CGVector(dx: 0.2, dy: 0.2)
push.pushDirection = vector
```

Continuous push behavior can be achieved by changing the *mode* in the above code property to *UIPushBehaviorMode.continuous*.

To change the point where force is applied, configure the behavior using the *setTargetOffsetFromCenter(_:for:)* method of the behavior object, specifying an offset relative to the center of the view. For example:

```
let offset = UIOffset(horizontal: 20, vertical: 20)
push.setTargetOffsetFromCenter(offset, for:view1)
```

In most cases, an off-center target for the pushing force will cause the item to rotate as it moves, as indicated in Figure 65-3:

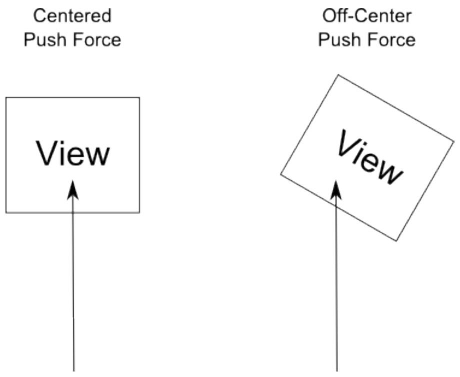

Figure 65-3

65.10 The UIDynamicItemBehavior Class

The UIDynamicItemBehavior class allows additional behavior characteristics to be defined that complement a number of the above primitive behaviors. This class can, for example, be used to define the density, resistance, and elasticity of dynamic items so that they do not move as far when subjected to an instantaneous push or bounce to a greater extent when involved in a collision. Dynamic items also can rotate by default. If rotation is not required for an item, this behavior can be turned off using a UIDynamicItemBehavior instance.

The behavioral properties of dynamic items that the UIDynamicItemBehavior class can govern are as follows:

- **allowsRotation** – Controls whether or not the item is permitted to rotate during animation.
- **angularResistence** – The amount by which the item resists rotation. The higher the value, the faster the item will stop rotating.
- **density** – The mass of the item.
- **elasticity** – The amount of elasticity an item will exhibit when involved in a collision. The greater the elasticity, the more the item will bounce.

- **friction** – The resistance exhibited by an item when it slides against another item.
- **resistance** – The overall resistance that the item exhibits in response to behavioral influences. The greater the value, the sooner the item will come to a complete stop during animation.

In addition, the class includes the following methods that may be used to increase or decrease the angular or linear velocity of a specified dynamic item:

- **angularVelocity(for:)** – Increases or decreases the angular velocity of the specified item. Velocity is specified in radians per second, where a negative value reduces the angular velocity.
- **linearVelocity(for:)** – Increases or decreases the linear velocity of the specified item. Velocity is specified in points per second, where a negative value reduces the velocity.

The following code example creates a new UIDynamicItemBehavior instance and uses it to set resistance and elasticity for two views before adding the behavior to the dynamic animator instance:

```
let behavior = UIDynamicItemBehavior(items: [view1, view2])
behavior.elasticity = 0.2
behavior.resistance = 0.5
animator?.addBehavior(behavior)
```

65.11 Combining Behaviors to Create a Custom Behavior

Multiple behaviors may be combined to create a single custom behavior using an instance of the UIDynamicBehavior class. The first step is to create and initialize each of the behavior objects. An instance of the UIDynamicBehavior class is then created, and each behavior is added to it via calls to the *addChildBehavior* method. Once created, only the UIDynamicBehavior instance needs to be added to the dynamic animator. For example:

```
// Create multiple behavior objects here

let customBehavior = UIDynamicBehavior()

customBehavior.addChildBehavior(behavior)
customBehavior.addChildBehavior(attachment)
customBehavior.addChildBehavior(gravity)
customBehavior.addChildBehavior(push)

animator?.addBehavior(customBehavior)
```

65.12 Summary

UIKit Dynamics provides a new way to bridge the gap between user interaction with an iOS device and corresponding animation within an app user interface. UIKit Dynamics takes a novel approach to animation by allowing view items to be configured such that they behave in much the same way as physical objects in the real world. This chapter has covered an overview of the basic concepts behind UIKit Dynamics and provided some details on how such behavior is implemented in terms of coding. The next chapter will work through a tutorial demonstrating many of these concepts.