Introduction to iOS 7

Overview

iOS 7 is a major update to iOS. It introduces a completely new user interface design that puts focus on content rather than application chrome. Alongside the visual changes, iOS 7 adds a plethora of new APIs to create richer interactions and experiences. This document surveys the new technologies introduced with iOS 7, and serves as a starting point for further exploration.

View Controller Transitions

UIKit adds support for customizing the animated transition that occurs when presenting view controllers. This support is included with built-in controllers, as well as any custom controllers that inherit directly from `UIViewController`. Additionally, `UICollectionViewController` takes advantage of controller transition customization to leverage the animated transitions in collection view layouts.

Custom Transitions

The animated transition between view controllers in iOS 7 is fully customizable. `UIViewController` now includes a `TransitioningDelegate` property that provides a custom animator class to the system when a transition occurs.

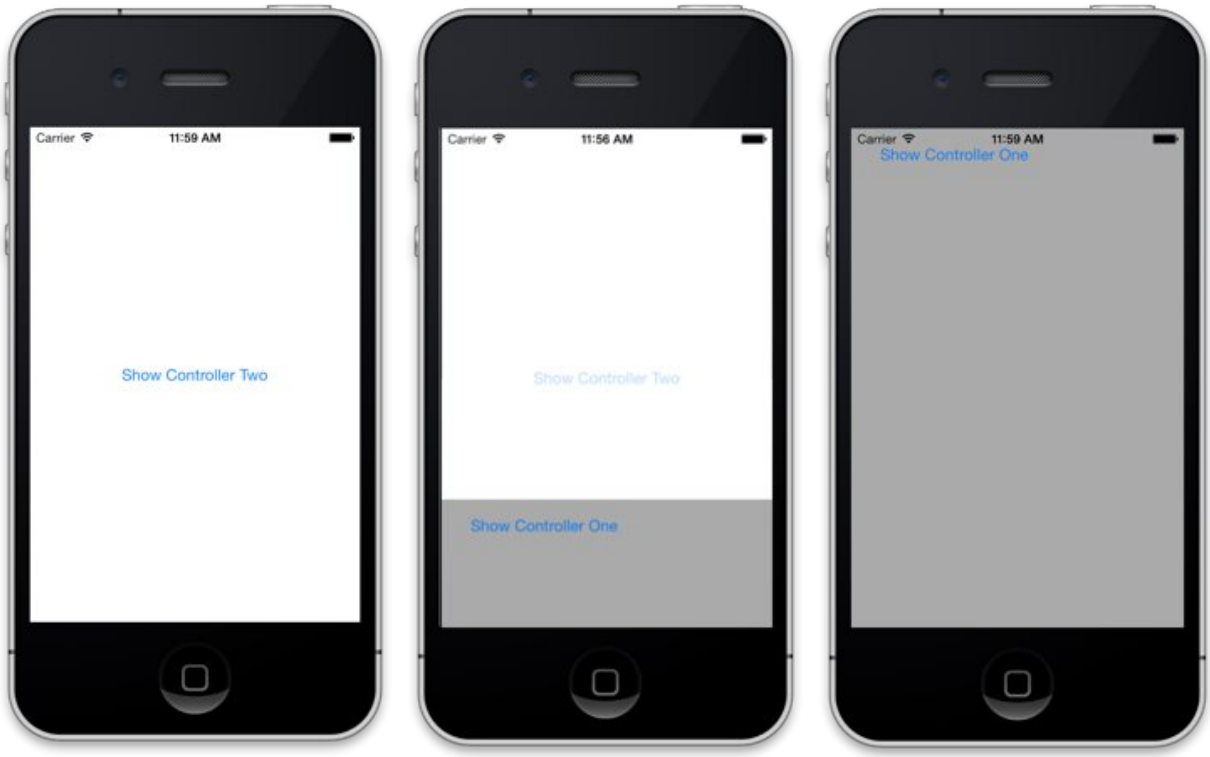
To use a custom transition with `PresentViewController`:

1. Set the `ModalPresentationStyle` to `UIModalPresentationStyle.Custom` on the controller to be presented.
2. Implement `UIViewControllerTransitioningDelegate` to create an animator class, which is an instance of `UIViewControllerAnimatedTransitioning` .
3. Set the `TransitioningDelegate` property to an instance of `UIViewControllerTransitioningDelegate` , also on the controller to be presented.
4. Present the view controller.

For example, the following code presents a view controller of type `ControllerTwo` - a `UIViewController` subclass:

```
showTwo.TouchUpInside += (object sender, EventArgs e) => {

    controllerTwo = new ControllerTwo ();

    this.PresentViewController (controllerTwo, true, null);
};
```

Running the app and tapping the button causes the default animation of the second controller's view to animate in from the bottom, as shown below:



However, setting the `ModalPresentationStyle` and `TransitioningDelegate` results in a custom animation for the transition:

```
showTwo.TouchUpInside += (object sender, EventArgs e) => {

    controllerTwo = new ControllerTwo () {
        ModalPresentationStyle = UIModalPresentationStyle.Custom;
        };

    transitioningDelegate = new TransitioningDelegate ();
    controllerTwo.TransitioningDelegate = transitioningDelegate;
```

```
    this.PresentViewController (controllerTwo, true, null);
};
```

The `TransitioningDelegate` is responsible for creating an instance of the `UIViewControllerAnimatedTransitioning` subclass - called `CustomAnimator` in the example below:

```
public class TransitioningDelegate : UIViewControllerTransitioningDelegate
{
    CustomTransitionAnimator animator;

    public override IUIViewControllerAnimatedTransitioning PresentingController
(UIViewController presented, UIViewController presenting, UIViewController
source)
    {
        animator = new CustomTransitionAnimator ();
        return animator;
    }
}
```

When the transition takes place, the system creates an instance of `IUIViewControllerContextTransitioning`, which it passed to the animator's methods. `IUIViewControllerContextTransitioning` contains the `ContainerView` where the animation occurs, as well as the view controller initiating the transition and the view controller being transitioned to.

The `UIViewControllerAnimatedTransitioning` class handles the actual animation. Two methods must be implemented:

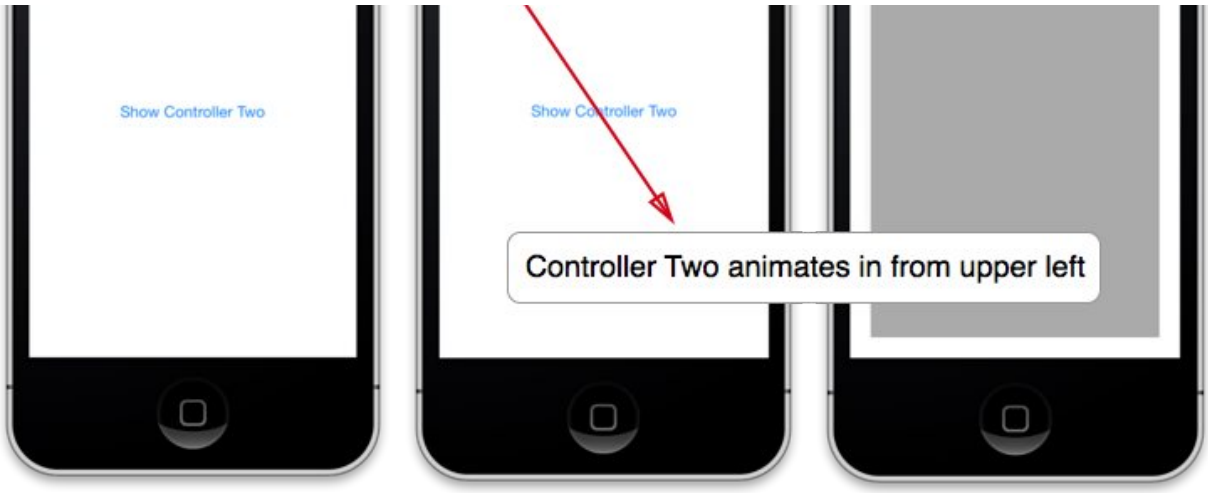
1. `TransitionDuration` – returns the duration of the animation in seconds.
2. `AnimateTransition` – performs the actual animation.

For example, the following class implements `UIViewControllerAnimatedTransitioning` to animate the frame of the controller's view:

```
public class CustomTransitionAnimator : UIViewControllerAnimatedTransitioning
{
    public CustomTransitionAnimator ()
    {
    }
```

```
    public override double TransitionDuration
(IUIViewControllerContextTransitioning transitionContext)
    {
        return 1.0;
    }

    public override void AnimateTransition
(IUIViewControllerContextTransitioning transitionContext)
        {
            var inView = transitionContext.ContainerView;
            var toVC = transitionContext.GetViewControllerForKey
(UITransitionContext.ToViewControllerKey);
            var toView = toVC.View;

            inView.AddSubview (toView);

            var frame = toView.Frame;
            toView.Frame = CGRect.Empty;

            UIView.Animate (TransitionDuration (transitionContext), () => {
                toView.Frame = new CGRect (20, 20, frame.Width - 40,
frame.Height - 40);
            }, () => {
                transitionContext.CompleteTransition (true);
            });
        }
}
```

Now, when the button is tapped, the animation implemented in the `UIViewControllerAnimatedTransitioning` class is used:



Collection View Transitions

Collection Views have built-in support for creating animated transitions:

- **Navigation Controllers** – The animated transition between two `UICollectionViewController` instances can optionally be handled automatically when a `UINavigationController` manages them.
- **Transition Layout** – A new `UICollectionViewTransitionLayout` class allows interactive transitioning between layouts.

Navigation Controller Transitions

When used within a navigation controller, a `UICollectionViewController` includes support for animated transitions between controllers. This support is built-in and requires only a few simple steps to implement:

1. Set `UseLayoutToLayoutNavigationTransitions` to `false` on a `UICollectionViewController`.
2. Add an instance of the `UICollectionViewController` to the root of the navigation controller's stack.
3. Create a second `UICollectionViewController` and set its `UseLayoutToLayoutNavigtionTransitions` property to `true`.
4. Push the second `UICollectionViewController` onto the navigation controller's stack.

The following code adds a `UICollectionViewController` subclass named `ImagesCollectionViewController` to the root of a navigation controller's stack, with the

`UseLayoutToLayoutNavigationTransitions` property set to `false`:

```
UIWindow window;
ImagesCollectionViewController viewController;
UICollectionViewFlowLayout layout;
UINavigationController navController;

public override bool FinishedLaunching (UIApplication app, NSDictionary
options)
{
    window = new UIWindow (UIScreen.MainScreen.Bounds);

    // create and initialize a UICollectionViewFlowLayout
    layout = new UICollectionViewFlowLayout (){
        SectionInset = new UIEdgeInsets (10,5,10,5),
        MinimumInteritemSpacing = 5,
        MinimumLineSpacing = 5,
        ItemSize = new CGSize (100, 100)
    };

    viewController = new ImagesCollectionViewController (layout) {
            UseLayoutToLayoutNavigationTransitions = false;
        }

    navController = new UINavigationController (viewController);

    window.RootViewController = navController;
    window.MakeKeyAndVisible ();

    return true;
}
```

When an item is selected, a second instance of the `ImagesController` is created, only this time using a different layout class. For this controller, `UseLayoutToLayoutNavigtionTransitions` is set to `true`, as shown below:

```
CircleLayout circleLayout;
ImagesCollectionViewController controller2;
```

```
...

public override void ItemSelected (UICollectionView collectionView, NSIndexPath
indexPath)
{
    // UseLayoutToLayoutNavigationTransitions when item is selected
        circleLayout = new CircleLayout (Monkeys.Instance.Count){
                ItemSize = new CGSize (100, 100)
            };

    controller2 = new ImagesCollectionViewController (circleLayout) {
        UseLayoutToLayoutNavigationTransitions = true;
        }

    NavigationController.PushViewController (controller2, true);
}
```

The `UseLayoutToLayoutNavigationTransitions` property must be set prior to adding the controller to the navigation stack. With this property set, the normal horizontal sliding transition is replaced with an animated transition between the layouts of the two controllers, as illustrated below:

Transition Layout

In addition to layout transition support within navigation controllers, a new layout called `UICollectionViewTransitionLayout` is now available. This layout class allows interactive control during the layout transition process, by allowing the `TransitionProgress` to be set from code. `UICollectionViewTransitionLayout` is different from - and not a replacement for - the `SetCollectionViewLayout` method from iOS 6 that caused an animated layout transition to occur. That method did not provide built-in support for controlling the progress of the animated transition.

`UICollectionViewTransitionLayout` allows, for example, a gesture recognizer to be configured to control the transition between layouts in response to user interaction, by managing the original layout as well as the intended layout to transition to.

The steps to implement an interactive transition within a gesture recognizer using `UICollectionViewTransitionLayout` are as follows:

1. Create a gesture recognizer.
2. Call the `StartInteractiveTransition` method of the `UICollectionView` , passing it the target layout and a completion handler.
3. Set the `TransitionProgress` property of the `UICollectionViewTransitionLayout` instance returned from the `StartInteractiveTransition` method.
4. Invalidate the layout.
5. Call the `FinishInteractiveTransition` method of the `UICollectionView` to complete the transition or the `CancelInteractiveTransition` method to cancel it. `FinishInteractiveTransition` causes the animation to complete its transition to the target layout, whereas `CancelInteractiveTransition` results in the animation returning to the original layout.
6. Handle the transition completion in the completion handler of the `StartInteractiveTransition` method.
7. Add the gesture recognizer to the collection view.

The following code implements an interactive layout transition within a pinch gesture recognizer:

```
imagesController = new ImagesCollectionViewController (flowLayout);

nfloat sf = 0.4f;
UICollectionViewTransitionLayout trLayout = null;
UICollectionViewLayout nextLayout;
```

```
pinch = new UIPinchGestureRecognizer (g => {

    var progress = Math.Abs(1.0f -  g.Scale)/sf;

    if(trLayout == null){
        if(imagesController.CollectionView.CollectionViewLayout is
CircleLayout)
            nextLayout = flowLayout;
        else
            nextLayout = circleLayout;

        trLayout = imagesController.CollectionView.StartInteractiveTransition
(nextLayout, (completed, finished) => {
            Console.WriteLine ("transition completed");
            trLayout = null;
        });
    }

    trLayout.TransitionProgress = (nfloat)progress;

    imagesController.CollectionView.CollectionViewLayout.InvalidateLayout ();

    if(g.State == UIGestureRecognizerState.Ended){
        if (trLayout.TransitionProgress > 0.5f)
            imagesController.CollectionView.FinishInteractiveTransition ();
        else
            imagesController.CollectionView.CancelInteractiveTransition ();
    }

});

imagesController.CollectionView.AddGestureRecognizer (pinch);
```

As the user pinches the collection view, the `TransitionProgress` is set relative to the scale of the pinch. In this implementation, if the user ends the pinch before the transition is 50% completed, the transition is cancelled. Otherwise, the transition is finished.

The following screenshots illustrate the transition between the layouts as the user pinches the collection view:

UIView Animation Enhancements

iOS 7 augments the animation support in UIKit, allowing applications to do things that previously required dropping directly into the Core Animation framework. For example, `UIView` can now perform spring animations as well as keyframe animations, which previously a `CAKeyframeAnimation` applied to a `CALayer`.

Spring Animations

`UIView` now supports animating property changes with a spring effect. To add this, call either the `AnimateNotify` or `AnimateNotifyAsync` method, passing in values for the spring damping ratio and the initial spring velocity, as described below:

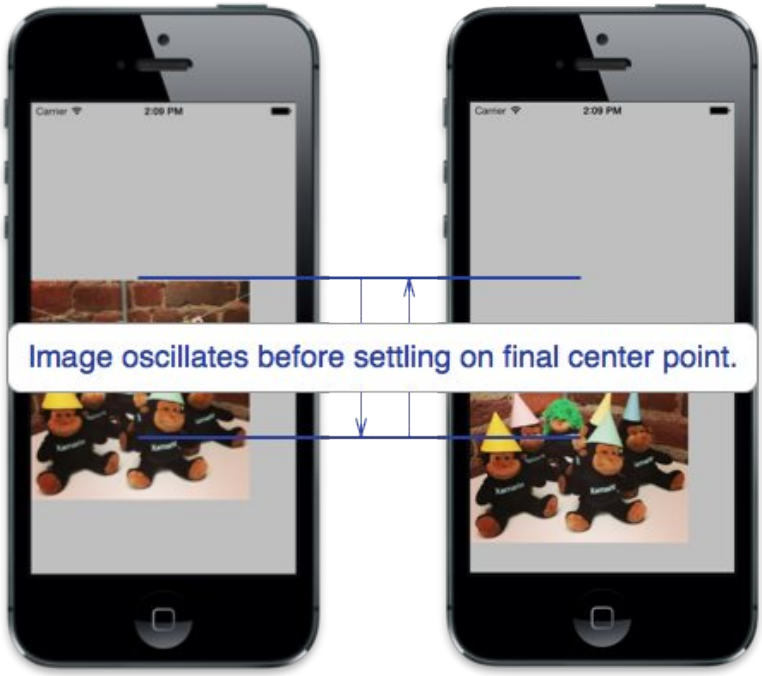
- `springWithDampingRatio` – A value between 0 and 1, where the oscillation increases for smaller value.
- `initialSpringVelocity` – The initial spring velocity as a percentage of the total animation distance per second.

The following code produces a spring effect when the image view's center changes:

```
void AnimateWithSpring ()
{
    float springDampingRatio = 0.25f;
    float initialSpringVelocity = 1.0f;

    UIView.AnimateNotify (3.0, 0.0, springDampingRatio, initialSpringVelocity,
0, () => {

        imageView.Center = new CGPoint (imageView.Center.X, 400);

    }, null);
}
```

This spring effect causes the image view to appear to bounce as it completes its animation to a new center

location, as illustrated below:



Keyframe Animations

The `UIView` class now includes the `AnimateWithKeyframes` method for creating keyframe animations on a `UIView`. This method is similar to other `UIView` animation methods, except that an additional `NSAction` is passed as a parameter to include the keyframes. Within the `NSAction`, keyframes are added by calling `UIView.AddKeyframeWithRelativeStartTime`.

For example, the following code snippet creates a keyframe animation to animate a view's center as well as to rotate the view:

```
void AnimateViewWithKeyframes ()
{
    var initialTransform = imageView.Transform;
    var initialCeneter = imageView.Center;

    // can now use keyframes directly on UIView without needing to drop
directly into Core Animation

    UIView.AnimateKeyframes (2.0, 0,
```

```
UIViewKeyframeAnimationOptions.Autoreverse, () => {
        UIView.AddKeyframeWithRelativeStartTime (0.0, 0.5, () => {
            imageView.Center = new CGPoint (200, 200);
        });

        UIView.AddKeyframeWithRelativeStartTime (0.5, 0.5, () => {
            imageView.Transform = CGAffineTransform.MakeRotation
((float)Math.PI / 2);
        });
    }, (finished) => {
        imageView.Center = initialCeneter;
        imageView.Transform = initialTransform;

        AnimateWithSpring ();
    });
}
```

The first two parameters to the `AddKeyframeWithRelativeStartTime` method specify the start time and duration of the keyframe, respectively, as a percentage of the overall animation length. The example above results in the image view animating to its new center over the first second, followed by rotating 90 degrees over the next second. Since the animation specifies `UIViewKeyframeAnimationOptions.Autoreverse` as an option, both keyframes animate in reverse as well. Finally, the final values are set to the initial state in the completion handler.

The screenshots below illustrates the combined animation through the keyframes:

UIKit Dynamics

UIKit Dynamics is a new set of APIs in UIKit that allow applications to create animated interactions based on physics. UIKit Dynamics encapsulates a 2D physics engine to make this possible.

The API is declarative in nature. You declare how the physics interactions behave by creating objects - called *behaviors* - to express physics concepts such as gravity, collisions, springs, etc. Then you attach the behavior(s) to another object, called a *dynamic animator*, which encapsulates a view. The dynamic animator takes cares of applying the declared physics behaviors to *dynamic items* - items that implement `IUIDynamicItem`, such as a `UIView`.

There are several different primitive behaviors available to trigger complex interactions, including:

- `UIAttachmentBehavior` – Attaches two dynamic items such that they move together, or attaches a dynamic item to an attachment point.
- `UICollisionBehavior` – Allows dynamic items to participate in collisions.
- `UIDynamicItemBehavior` – Specifies a general set of properties to apply to dynamic items, such as elasticity, density and friction.
- `UIGravityBehavior` - Applies gravity to a dynamic item, causing items to accelerate in the gravitational direction.
- `UIPushBehavior` – Applies force to a dynamic item.
- `UISnapBehavior` – Allows a dynamic item to snap to a position with a spring effect.

Although there are many primitives, the general process for adding physics-based interactions to a view using UIKit Dynamics is consistent across behaviors:

1. Create a dynamic animator.
2. Create behavior(s).
3. Add behaviors to the dynamic animator.

Dynamics Example

Let's look at an example that adds gravity and a collision boundary to a `UIView`.

UIGravityBehavior

Adding gravity to an image view follows the 3 steps outlined above.

We'll work in the `ViewDidLoad` method for this example. First, add a `UIImageView` instance as follows:

```
image = UIImage.FromFile ("monkeys.jpg");

imageView = new UIImageView (new CGRect (new CGPoint (View.Center.X -
image.Size.Width / 2, 0), image.Size)) {
                    Image =  image
                }

View.AddSubview (imageView);
```

This creates an image view centered at the top edge of the screen. To make the image "fall" with gravity, create an instance of a `UIDynamicAnimator`:

```
dynAnimator = new UIDynamicAnimator (this.View);
```

The `UIDynamicAnimator` takes an instance of a reference `UIView` or a `UICollectionViewLayout`, which contains the items that will be animated per the attached behavior(s).

Next, create a `UIGravityBehavior` instance. You can pass one or more objects implementing the `IUIDynamicItem`, like a `UIView`:

```
var gravity = new UIGravityBehavior (dynItems);
```

The behavior is passed an array of `IUIDynamicItem`, which in this case contains the single `UIImageView` instance we are animating.

Finally, add the behavior to the dynamic animator:

```
dynAnimator.AddBehavior (gravity);
```

This results in the image animating downward with gravity, as illustrated below:

Since there is nothing constraining the boundaries of the screen, the image view simply falls off the bottom. To constrain the view so that the image collides with the edges of the screen, we can add a `UICollisionBehavior`. We'll cover this in the next section.

UICollisionBehavior

We'll begin by creating a `UICollisionBehavior` and adding it to the dynamic animator, just like we did for the `UIGravityBehavior`.

Modify the code to include the `UICollisionBehavior`:

```
using (image = UIImage.FromFile ("monkeys.jpg")) {

    imageView = new UIImageView (new CGRect (new CGPoint (View.Center.X -
image.Size.Width / 2, 0), image.Size)) {
        Image =  image
    };

    View.AddSubview (imageView);

    // 1. create the dynamic animator
    dynAnimator = new UIDynamicAnimator (this.View);

    // 2. create behavior(s)
    var gravity = new UIGravityBehavior (imageView);
    var collision = new UICollisionBehavior (imageView) {
        TranslatesReferenceBoundsIntoBoundary = true
    };

    // 3. add behaviors(s) to the dynamic animator
    dynAnimator.AddBehaviors (gravity, collision);
}
```

The `UICollisionBehavior` has a property called `TranslatesReferenceBoundsIntoBoundry`. Setting this to `true` causes the reference view's bounds to be used as a collision boundary.

Now, when the image animates downward with gravity, it bounces slightly off the bottom of the screen before settling to rest there, as shown below:

UIDynamicItemBehavior

We can further control the behavior of the falling image view with additional behaviors. For example, we could add a `UIDynamicItemBehavior` to increase the elasticity, causing the image view to bounce more when it collides with the bottom of the screen.

Adding a `UIDynamicItemBehavior` follows the same steps as with the other behaviors. First create the behavior:

```
var dynBehavior = new UIDynamicItemBehavior (dynItems) {
    Elasticity = 0.7f
};
```

Then, add the behavior to the dynamic animator:

```
dynAnimator.AddBehavior (dynBehavior);
```

With this behavior in place, the image view bounces more when it collides with the boundary.

General User Interface Changes

In addition to the new UIKit APIs such as UIKit Dynamics, Controller transitions, and enhanced UIView animations described above, iOS 7 introduces a variety of visual changes to the UI, and related API changes for various views and controls. For more information see the iOS 7 User Interface Overview.

Text Kit

Text Kit is a new API that offers powerful text layout and rendering features. It is built on top of the low level Core Text framework, but is much easier to use than Core Text.

To make the features of Text Kit available to standard controls, several iOS text controls have been re-implemented to use Text Kit, including:

- UITextView
- UITextField
- UILabel

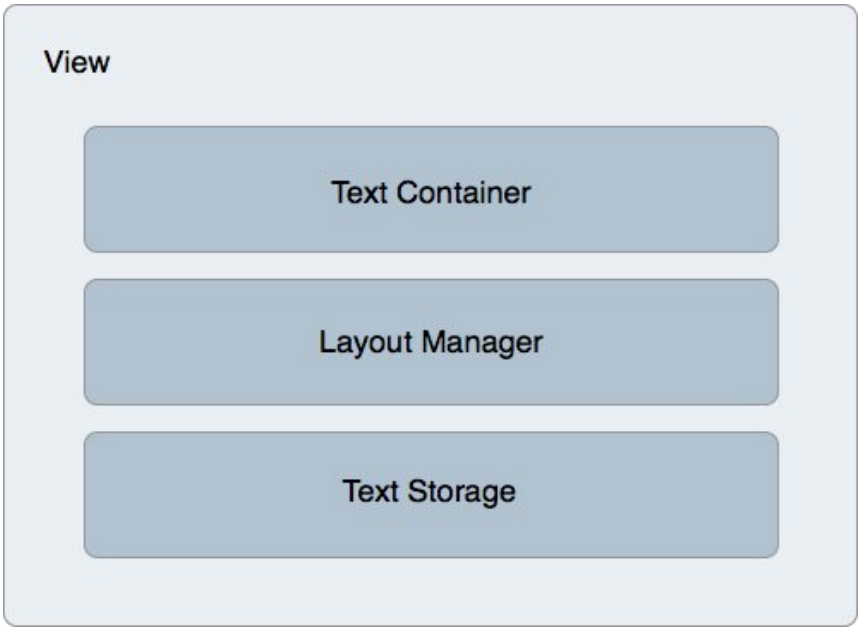
Architecture

Text Kit provides a layered architecture that separates the text storage from the layout and display, including the following classes:

- `NSTextContainer` – Provides the coordinate system and geometry that is used to layout text.
- `NSLayoutManager` – Lays out text by turning text into glyphs.
- `NSTextStorage` – Holds the text data, as well as handles batch text property updates. Any batch updates are handed to the layout manager for the actual processing of the changes, such as recalculating the layout and redrawing the text.

These three classes are applied to a view that renders text. The built-in text handling views, such as `UITextView`, `UITextField`, and `UILabel` already have them set, but you can create and apply them to any `UIView` instance as well.

The following figure illustrates this architecture:



Text Storage and Attributes

The `NSTextStorage` class holds the text that is displayed by a view. It also communicates any changes to the text - such as changes to characters or their attributes - to the layout manager for display. `NSTextStorage` inherits from `MSMutableAttributed` string, allowing changes to text attributes to be specified in batches between `BeginEditing` and `EndEditing` calls.

For example, the following code snippet specifies a change to the foreground and background colors, respectively, and targets particular ranges:

```
textView.TextStorage.BeginEditing ();
textView.TextStorage.AddAttribute(UIStringAttributeKey.ForegroundColor,
UIColor.Green, new NSRange(200, 400));
textView.TextStorage.AddAttribute(UIStringAttributeKey.BackgroundColor,
UIColor.Black, new NSRange(210, 300));
textView.TextStorage.EndEditing ();
```

After `EndEditing` is called, the changes are sent to the layout manager, which in turn performs any necessary layout and rendering calculations for the text to be displayed in the view.

The following screenshot shows the text with the specified attributes displayed in the text view:

Layout with Exclusion Path

Text Kit also supports layout, and allows for complex scenarios such as multi-column text and flowing text around specified paths called *exclusion paths*. Exclusion paths are applied to the text container, which modifies the geometry of the text layout, causing the text to flow around the specified paths.

Adding an exclusion path requires setting the `ExclusionPaths` property on the layout manager. Setting this property causes the layout manager to invalidate the text layout and flow the text around the exclusion path.

Exclusion based on a CGPath

Consider the following `UITextView` subclass implementation:

```
public class ExclusionPathView : UITextView
{
    CGPath exclusionPath;
    CGPoint initialPoint;
    CGPoint latestPoint;
```

```
UIBezierPath bezierPath;

public ExclusionPathView (string text)
{
    Text = text;
    ContentInset = new UIEdgeInsets (20, 0, 0, 0);
    BackgroundColor = UIColor.White;
    exclusionPath = new CGPath ();
    bezierPath = UIBezierPath.Create ();

    LayoutManager.AllowsNonContiguousLayout = false;
}

public override void TouchesBegan (NSSet touches, UIEvent evt)
{
    base.TouchesBegan (touches, evt);

    var touch = touches.AnyObject as UITouch;

    if (touch != null) {
        initialPoint = touch.LocationInView (this);
    }
}

public override void TouchesMoved (NSSet touches, UIEvent evt)
{
    base.TouchesMoved (touches, evt);

    UITouch touch = touches.AnyObject as UITouch;

    if (touch != null) {
        latestPoint = touch.LocationInView (this);
        SetNeedsDisplay ();
    }
}

public override void TouchesEnded (NSSet touches, UIEvent evt)
```

```
    {
        base.TouchesEnded (touches, evt);

        bezierPath.CGPath = exclusionPath;
        TextContainer.ExclusionPaths = new UIBezierPath[] { bezierPath };
    }

    public override void Draw (CGRect rect)
    {
        base.Draw (rect);

        if (!initialPoint.IsEmpty) {

            using (var g = UIGraphics.GetCurrentContext ()) {

                g.SetLineWidth (4);
                UIColor.Blue.SetStroke ();

                if (exclusionPath.IsEmpty) {
                    exclusionPath.AddLines (new CGPoint[] { initialPoint,
latestPoint });
                } else {
                    exclusionPath.AddLineToPoint (latestPoint);
                }

                g.AddPath (exclusionPath);
                g.DrawPath (CGPathDrawingMode.Stroke);
            }
        }
    }
}
```

This code adds support for drawing on the text view using Core Graphics. Since the `UITextView` class is now built to use Text Kit for its text rendering and layout, it can use all the features of Text Kit, such as setting exclusion paths.

Note: This example subclasses `UITextView` to add touch drawing support. Subclassing `UITextView` isn't necessary to get the features of Text Kit.

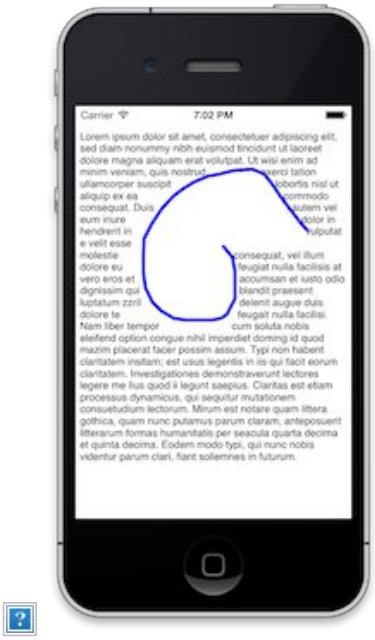
After the user draws on the text view, the drawn `CGPath` is applied to a `UIBezierPath` instance by setting the `UIBezierPath.CGPath` property:

```
bezierPath.CGPath = exclusionPath;
```

Updating the following line of code makes the text layout update around the path:

```
TextContainer.ExclusionPaths = new UIBezierPath[] { bezierPath };
```

The following screenshot illustrates how the text layout changes to flow around the drawn path:



Notice that the layout manager's `AllowsNonContiguousLayout` property is set to false in this case. This causes the layout to be recalculated for all cases where the text changes. Setting this to true may benefit performance by avoiding a full-layout refresh, especially in the case of large documents. However, setting `AllowsNonContiguousLayout` to true would prevent the exclusion path from updating the layout in some circumstances - for example, if text is entered at runtime without a trailing carriage return prior to the path being set.

Multitasking

iOS 7 changes when and how background work is performed. Task completion in iOS 7 no longer keeps applications awake when tasks are running in the background, and applications are woken for background processing in a non-contiguous manner. iOS 7 also adds three new APIs for updating applications with

new content in the background:

- Background Fetch – Allows applications to update content in the background at regular intervals.
- Remote Notifications - Allows applications to update content when receiving a push notification. The notifications can be either silent or can display a banner on the lock screen.
- Background Transfer Service – Allows uploading and downloading of data, such as large files, without a fixed time limit.

For more details about the new multitasking capabilities, see the iOS sections of the Xamarin Backgrounding guide.

Summary

This article covers several major new additions to iOS. First, it shows how to add custom transitions to View Controllers. Then, it shows how to use transitions in collection views, both from within a navigation controller, as well as interactively between collection views. Next, it introduces several enhancements made to UIView animations, showing how applications use UIKit for things that previously required programming directly against Core Animation. Finally, new UIKit Dynamics API, which brings a physics engine to UIKit, is introduced alongside the rich text support now available in the Text Kit framework.