Sample Code

Display text with a custom layout

Lay out text in a custom-shaped container and apply glyph substitutions.

Download

iOS 13.0+ | iPadOS 13.0+ | Xcode 12.0+

Overview

Some apps, such as book and magazine readers, text editors, and games, may need to lay out their text in a way that better fits their app style. TextKit provides a set of APIs for these apps to implement a custom text layout. This sample demonstrates how to use the APIs to display text in a circular container and in a two-column container, how to set up an exclusive area for a text container, and how to substitute a glyph without changing the text storage.

Implement a custom-shaped text container

When laying out a line of text, TextKit calls the `lineFragmentRect(forProposedRect:at:writingDirection:remaining:)` method from `NSTextContainer` to determine the position and size of the line, which TextKit calls a *line fragment rectangle*. By creating a subclass of `NSTextContainer` to return a custom line fragment rectangle in the method, apps can implement a custom-shaped text container.

This sample uses the `CircleTextContainer` class to implement a circular text container. To calculate a line fragment rectangle that fits in the inscribed circle of the container's bounds, the class calls the implementation of `super` to retrieve the default rectangle, then adjusts its `origin.x` and `width` according to the current line origin and container size.

```
override func lineFragmentRect(forProposedRect proposedRect: CGRect,
                               at characterIndex: Int,
                               writingDirection baseWritingDirection: NSWritingDirec
```

```
                                remaining remainingRect: UnsafeMutablePointer<CGRect>
    let rect = super.lineFragmentRect(forProposedRect: proposedRect,
                                      at: characterIndex,
                                      writingDirection: baseWritingDirection,
                                      remaining: remainingRect)
    let containerWidth = Float(size.width), containerHeight = Float(size.height)

    let diameter = fminf(containerWidth, containerHeight)
    let radius = diameter / 2.0

    // Vertical distance from the line center to the container center.
    let yDistance = fabsf(Float(rect.origin.y + rect.size.height / 2.0) - radius)
    // The new line width.
    let width = (yDistance < radius) ? 2.0 * sqrt(radius * radius - yDistance * yDis
    // Horizontal distance from rect.origin.x to the starting point of the line.
    let xOffset = (containerWidth > diameter) ? (containerWidth - diameter) / 2.0 :
    // The starting x of the line.
    let xPosition = CGFloat(xOffset + Float(rect.origin.x) + radius - width / 2.0)
    return CGRect(x: xPosition, y: CGFloat(rect.origin.y), width: CGFloat(width), he
}
```

Lay out text with a custom text container

To lay out text with a custom text container, apps simply set up a text view with the container, and let TextKit do the rest. The `init(frame:textContainer:)` method from `UITextView` serves this purpose, and this sample uses it to create a `UITextView` instance with its circular text container.

```
let textContainer = CircleTextContainer(size: .zero)
textContainer.widthTracksTextView = true

let layoutManager = NSLayoutManager()
layoutManager.addTextContainer(textContainer)
textStorage.addLayoutManager(layoutManager)

textView = UITextView(frame: CGRect.zero, textContainer: textContainer)
```

With this configuration, the TextKit class that coordinates the layout and display of characters, `NSLayoutManager`, automatically uses the line fragment rectangles that `CircleTextContainer` returns to lay out the text.

`NSLayoutManager` supports laying out text in multiple text containers, so implementing a two-column layout is as easy as adding a second text container to the layout manager, as the code below shows:

```
let firstTextContainer = NSTextContainer()
firstTextContainer.widthTracksTextView = true
firstTextContainer.heightTracksTextView = true

let secondTextContainer = NSTextContainer()
secondTextContainer.widthTracksTextView = true
secondTextContainer.heightTracksTextView = true
secondTextContainer.lineBreakMode = .byTruncatingTail

let layoutManager = NSLayoutManager()
layoutManager.addTextContainer(firstTextContainer)
layoutManager.addTextContainer(secondTextContainer)

textStorage.addLayoutManager(layoutManager)

let firstTextView = UITextView(frame: .zero, textContainer: firstTextContainer)
firstTextView.isScrollEnabled = false
view.addSubview(firstTextView)

let secondTextView = UITextView(frame: .zero, textContainer: secondTextContainer)
secondTextView.isScrollEnabled = false
view.addSubview(secondTextView)
```

Note that after adding a second text container to the layout manager, the text views become uneditable and unselectable.

Reserve an area in a text container

To create an appealing UI, some apps may have their text wrap around a certain shape. They can achieve that by using the `exclusionPaths` property from `NSTextContainer` to reserve an exclusive area for the shape in a text container.

This sample uses the following code to set up an exclusive area where `translatedCirclePath` is a `UIBezierPath` instance using the text container's coordinate system.

```
textView.textContainer.exclusionPaths = [translatedCirclePath]
```

Substitute glyphs without changing the text storage

When a text container doesn't have enough space to display text, apps may need a way to indicate that the container has additional, or *overflow*, text. A standard text container can use the `lineBreakMode` property from `NSTextContainer` to add an ending ellipsis that helps handle this situation, but the property doesn't completely support custom-shaped containers like `CircleTextContainer`.

To show an overflow indicator in its circular text container, as well as to demonstrate TextKit's capability of glyph substitutions and layout adjustment, this sample substitutes the glyphs of the container's ending characters with an ellipsis, and makes it the container's last visible glyph.

The ending characters are the characters of the last words with a total width that is larger than an ellipsis. The characters have to be wider so that the text container has enough space to display the ellipsis. However, because they are wider, the text container has extra space after the glyph substitution, which can drag in more characters. To avoid showing text after the ellipsis, this sample implements a second glyph substitution by selecting the character next to the container's last word, and substituting it with a control glyph that uses the `whitespace` action. Using this substitution, the character becomes a whitespace with a width that is flexible. With this flexible space character, the sample can fill up the extra space and push the extra characters to the next line, and then move the line out of the text container.

The detailed implementation of the glyphs substitutions and the layout adjustment is as follows:

- Begin the glyph substitutions by invalidating the glyphs and layout of the ending characters.

```
layoutManager.invalidateGlyphs(forCharacterRange: endingWordsCharRange, changeInLeng
                               actualCharacterRange: nil)
layoutManager.invalidateLayout(forCharacterRange: endingWordsCharRange,
                               actualCharacterRange: nil)
```

- Implement the `layoutManager(_:shouldGenerateGlyphs:properties:characterIndexes:font:forGlyphRange:)` method from `NSLayoutManagerDelegate` to substitute the glyphs. TextKit calls this delegate method before storing the glyphs to give apps an opportunity to change the glyphs and their properties.

```
let ellipsisStartIndex = ellipsisIntersection.location
for index in ellipsisStartIndex..<ellipsisStartIndex + ellipsisIntersection.length {
    if index == ellipsisGlyphRange!.location {
        finalGlyphs[index - glyphRange.location] = myGlyphs[0]
    } else {
        finalProps[index - glyphRange.location] = .controlCharacter
    }
}
```

```
let flexibleSpaceStartIndex = flexibleSpaceIntersection.location
for index in  flexibleSpaceStartIndex..<flexibleSpaceStartIndex + flexibleSpaceInter
    finalProps[index - glyphRange.location] = .controlCharacter
}
```

- Implement the `layoutManager(_:shouldUse:forControlCharacterAt:)` method to return the `NSLayoutManager.ControlCharacterAction.whitespace` action for the flexible space character.

```
func layoutManager(_ layoutManager: NSLayoutManager, shouldUse action: NSLayoutManag
                   forControlCharacterAt charIndex: Int) -> NSLayoutManager.ControlC
    if let flexibleSpaceGlyphRange = self.flexibleSpaceGlyphRange,
        flexibleSpaceGlyphRange.contains(layoutManager.glyphIndexForCharacter(at: ch
        return .whitespace
    }
    return action
}
```

- Implement the `layoutManager(_:boundingBoxForControlGlyphAt:for:proposedLineFragment:glyphPosition:characterIndex:)` method to return a bounding box that can fill up the current line fragment rectangle.

```
func layoutManager(_ layoutManager: NSLayoutManager,
                   boundingBoxForControlGlyphAt glyphIndex: Int,
                   for textContainer: NSTextContainer,
                   proposedLineFragment proposedRect: CGRect,
                   glyphPosition: CGPoint,
                   characterIndex charIndex: Int) -> CGRect {
    guard let flexibleSpaceGlyphRange = self.flexibleSpaceGlyphRange,
        flexibleSpaceGlyphRange.contains(glyphIndex) else {
        return CGRect(x: glyphPosition.x, y: glyphPosition.y, width: 0, height: prop
    }
    let padding = textContainer.lineFragmentPadding * 2
    let width = proposedRect.width - (glyphPosition.x - proposedRect.minX) - padding
    let rect = CGRect(x: glyphPosition.x, y: glyphPosition.y, width: width, height:
    return rect
}
```

- Implement the `layoutManager(_:shouldSetLineFragmentRect:lineFragmentUsedRect:baselineOffset:in:forGlyphRange:)` method to move the extra line out of the text container.

```
func layoutManager(_ layoutManager: NSLayoutManager,
                   shouldSetLineFragmentRect lineFragmentRect: UnsafeMutablePointer<
                   lineFragmentUsedRect: UnsafeMutablePointer<CGRect>,
                   baselineOffset: UnsafeMutablePointer<CGFloat>,
                   in textContainer: NSTextContainer,
                   forGlyphRange glyphRange: NSRange) -> Bool {
    guard let ellipsisGlyphRange = self.ellipsisGlyphRange,
        glyphRange.location > ellipsisGlyphRange.location else {
            return false
    }
    let originX = textContainer.size.width
    lineFragmentRect.pointee.origin = CGPoint(x: originX, y: lineFragmentRect.pointe
    return true
}
```

See Also

Layout

{} Using TextKit 2 to interact with text

Interact with text by managing text selection and inserting custom text elements.

`class` NSTextLayoutManager

The primary class that you use to manage text layout and presentation for custom text displays.

`class` NSTextContainer

A region where text layout occurs.

`class` NSTextLayoutFragment

A class that represents the layout fragment typically corresponding to a rendering surface, such as a layer or view subclass.

`class` NSTextLineFragment

A class that represents a line fragment as a single textual layout and rendering unit inside a text layout fragment.

`class` NSTextViewportLayoutController

Manages the layout process inside the viewport interacting with its delegate.

protocol NSTextLayoutOrientationProvider

A set of methods that define the orientation of text for an object.