Wie funktioniert YYText?

Der obige Schatteneffekt wurde mit dem folgenden Code erzielt:

Man kann sehen, dass zuerst ein `YYTextShadow` erzeugt wurde, dann dem `yy_textShadow` der `attributedString` zugewiesen wurde, und anschließend die `attributedString` dem `YYLabel` zugewiesen wurde. Danach wurde das `YYLabel` zur Anzeige in eine `UIView` eingefügt. Beim Verfolgen von `yy_textShadow` stellt man fest, dass hauptsächlich der `textShadow` an das Attribut `NSAttributedString` gebunden wurde, wobei der Schlüssel `YYTextShadowAttributeName` und der Wert `textShadow` ist. Das bedeutet, dass der Schatten zunächst gespeichert und später verwendet wird. Mit Shift + Command + J kann man schnell zur Definitionsstelle springen:

Hier gibt es eine Funktion namens `addAttribute`, die in `NSAttributedString.h` definiert ist:

```
- (void)addAttribute:(NSString *)name value:(id)value range:(NSRange)range;
```

Die Beschreibung besagt, dass beliebige Schlüssel-Wert-Paare zugewiesen werden können. Die Definition von `YYTextShadowAttributeName` ist ein gewöhnlicher String, was bedeutet, dass zunächst die Schatteninformationen gespeichert und später verwendet werden. Lassen Sie uns global nach `YYTextShadowAttributeName` suchen.

Dann kommen wir zur Funktion `YYTextDrawShadow` in `YYTextLayout`:

`CGContextTranslateCTM` bedeutet, den Ursprungspunkt eines Kontexts zu verschieben, also

```
CGContextTranslateCTM(context, point.x, point.y);
```

Es bedeutet, dass der Zeichenkontext zum Punkt `point` verschoben werden soll. Zuerst sollten wir herausfinden, wo `YYTextDrawShadow` aufgerufen wird, und wir stellen fest, dass es in `drawInContext` aufgerufen wird.

In `drawInContext` wird zuerst der Rahmen des Blocks gezeichnet, gefolgt vom Hintergrundrahmen, Schatten, Unterstreichung, Text, Zusatzelementen, innerem Schatten, Durchstreichung, Textrahmen und Debug-Linien.

Wo genau wird also `drawInContext` verwendet? Man kann sehen, dass es einen Parameter `YYTextDebugOption` gibt, daher ist diese Funktion definitiv kein System-Callback, sondern wird innerhalb von YYText selbst aufgerufen.

Halten Sie **Ctrl + 1** gedrückt, um die Tastenkombination aufzurufen, und Sie werden feststellen, dass sie an vier Stellen verwendet wird.

drawInContext:size:debug ist immer noch ein eigener Aufruf von YYText, da der Typ von debug YYTextDebugOption * ist, was zu YY gehört. newAsyncTask scheint kein Systemaufruf zu sein, und das gleiche gilt für addAttachmentToView:layer:, daher handelt es sich höchstwahrscheinlich um drawRect:.

Tatsächlich, wenn man sich die schnelle Hilfe auf der rechten Seite ansieht, gibt es eine detaillierte Erklärung, und unter der Hilfe wird auch erwähnt, dass es in UIView definiert ist. Wenn man sich dann YYTextContainerView ansieht, erbt es von UIView.

Also verwendet YYLabel YYTextContainerView? Und lässt dann das System die drawRect:-Methode in YYTextContainerView aufrufen, um das Zeichnen durchzuführen?

Seltsam, YYLabel erbt von UIView. Daher sollte es in YYText zwei verschiedene Systeme geben! Ein System YYLabel und ein System YYTextView, ähnlich wie UILabel und UITextView. Dann schauen wir uns noch einmal den newAsyncDisplayTask von YYLabel an, den wir zuvor gesehen haben.

Lange, in der Mitte wird die Methode drawInContext aus YYTextLayout aufgerufen. newAsyncDisplayTask, wo wird das wiederum aufgerufen?

In der zweiten Zeile wurde es aufgerufen. Man kann es also einfach so verstehen, dass YYLabel asynchron verwendet wird, um den Text zu zeichnen. Und _displayAsync wird von dem oben genannten display aufgerufen. Wenn man sich die Dokumentation von display ansieht, steht dort, dass das System es zum richtigen Zeitpunkt aufruft, um den Inhalt des Layers zu aktualisieren, und man sollte es nicht direkt aufrufen. Wir können auch einen Breakpoint setzen.

Die Erklärung besagt, dass display innerhalb einer Transaktion von CALayer aufgerufen wird. Der Grund für die Verwendung einer Transaktion liegt wahrscheinlich darin, dass Änderungen in Batches vorgenommen werden sollen, um die Effizienz zu erhöhen. Es scheint nicht so, als ob es sich um eine Anforderung für ein Rollback wie in einer Datenbank handelt.

Die Systemdokumentation von display besagt auch, dass Sie diese Methode überschreiben können, um Ihr eigenes Zeichnen zu implementieren, wenn Sie möchten, dass Ihre Ebene anders gezeichnet wird.

Also haben wir eine einfache Idee. YYLabel überschreibt die display-Methode von UIView, um asynchron verschiedene Effekte wie Schatten zu zeichnen. Der Schatteneffekt wird zunächst in den Attributen des attributedText von YYLabel gespeichert und dann während des Zeichnens in der display-Methode wieder abgerufen. Beim Zeichnen wird das CoreGraphics-Framework des Systems verwendet.

Nachdem ich einige Gedanken sortiert habe, wird mir klar, was wirklich beeindruckend ist: Einerseits die Organisation all dieser Effekte und asynchronen Aufrufe, andererseits die fundierte

Beherrschung des zugrunde liegenden CoreGraphics-Frameworks. Nachdem ich also ein Verständnis für die Organisation des vorherigen Codes entwickelt habe, tauchen wir tiefer in das CoreGraphics-Framework ein. Schauen wir uns an, wie die Zeichnungen tatsächlich umgesetzt werden.

Lassen Sie uns noch einmal zu `YYTextDrawShadow` zurückkehren.

Hier umschließen `CGContextSaveGState` und `CGContextRestoreGState` einen Block von Zeichencode. `CGContextSaveGState` bedeutet, dass der aktuelle Zeichenzustand kopiert und auf den Zeichenstapel gelegt wird. Jeder Zeichenkontext verwaltet einen eigenen Zeichenstapel. Ich bin mir nicht sicher, wie genau der Stapel intern funktioniert. Vorerst verstehe ich es so, dass vor dem Zeichnen im Kontext `CGContextSaveGState` aufgerufen werden muss und nach dem Zeichnen `CGContextRestoreGState`, damit die Zeichnungen in der Mitte effektiv im Kontext erscheinen. `CGContextTranslateCTM` bewegt den Kontext an die entsprechende Position. Zuerst wird zu `point.x` und `point.y` bewegt, um an der entsprechenden Position zu zeichnen. Warum danach zu 0 und `size.height` bewegt wird, ist mir noch nicht klar, das werde ich später noch einmal überprüfen. Anschließend wird `lines` ausgelesen und eine `for`-Schleife ausgeführt.

`lines` □ □ □□ □ `YYTextLayout` □ □ `(YYTextLayout *)layoutWithContainer:(YYTextContainer *)container text:(NSAttributedString *)text range:(NSRange)range` □□□□□□□

Dann navigieren Sie zur Definition dieser Funktion:

Diese Funktion ist sehr lang, sie erstreckt sich von Zeile 367 bis 861, insgesamt 500 Zeilen Code! Wenn man den Anfang und das Ende betrachtet, wird deutlich, dass ihr Zweck darin besteht, diese Variablen zu erhalten. Wie wird `lines` erhalten?

Man kann sehen, dass in einer großen `for`-Schleife Zeile für Zeile (`line`) zu `lines` hinzugefügt wird. Aber wie wird `lineCount` ermittelt?

In Zeile 472 wird ein `framesetter`-Objekt erstellt, wobei der Parameter `text` ein `NSAttributedString` ist. Anschließend wird in dem `frameSetter`-Objekt ein `CTFrameRef` erzeugt, und aus diesem `CTFrameRef` werden die `lines` extrahiert. Was genau ist eine `line`? Setzen wir einen Breakpoint, um das zu untersuchen.

Es wurde festgestellt, dass `lineCount = 2` für das Wort `shadow` nicht die erwartete Anzahl der Buchstaben darstellt.

Also könnte man vermuten, dass der weiße `Shadow` insgesamt eine `line` ist, und der Schatten ebenfalls eine `line`?

In YYText, there are several examples, but only one effect is displayed, and the other code is commented out. I noticed something strange: for Shadow, `lineCount = 2`, and for Multi-

ple Shadows, `lineCount` is also 2. However, Multiple Shadows also has an inner shadow, so shouldn't it be 3 lines?

In der Apple-Dokumentation zu CTLine steht, dass CTLine eine Textzeile repräsentiert und ein CTLine-Objekt eine Gruppe von `glyph runs` enthält. Es handelt sich also einfach um die Anzahl der Zeilen! Wenn man sich den obigen Screenshot des Breakpoints ansieht, war der Wert von `shadow` 2, weil der Text `shadow\n\n` lautete. Die `\n\n` wurden absichtlich hinzugefügt, um die Darstellung zu verschönern:

Also ist `shadow\n\n` ein Text mit zwei Zeilen. CTLine ist das, was wir normalerweise als Zeile bezeichnen. Schauen wir uns nun unseren `lineCount` noch einmal an:

Hier erhalten wir das `CTLines`-Array, bestimmen die Anzahl der darin enthaltenen Elemente und wenn `lineCount` größer als 0 ist, ermitteln wir den Ursprungspunkt jeder Zeile. Gut, jetzt, da wir `lineCount` haben, schauen wir uns die `for`-Schleife an.

Aus dem `ctLines`-Array wird ein `CTLine`-Objekt extrahiert, woraus dann ein `YYTextLine`-Objekt erstellt wird, das anschließend dem `lines`-Array hinzugefügt wird. Danach werden einige Frame-Berechnungen für die `line` durchgeführt. Der Konstruktor von `YYTextLine` ist recht einfach und speichert zunächst die Position, ob es sich um vertikalen Text handelt, und das `CTLine`-Objekt:

Nachdem wir `lines` verstanden haben, kehren wir zu `YYTextDrawShadow` zurück:

Der Code ist jetzt einfacher. Zuerst wird die Anzahl der Zeilen ermittelt, dann wird jede Zeile durchlaufen, und anschließend wird das `GlyphRuns`-Array abgerufen, das ebenfalls durchlaufen wird. Ein `GlyphRun` kann als ein grafisches Element oder eine Zeichnungseinheit verstanden werden. Danach wird das `attributes`-Array daraus extrahiert, und mit unserem zuvor definierten `YYTextShadowAttributeName` wird der `shadow`-Wert abgerufen, den wir am Anfang zugewiesen haben. Schließlich wird der Schatten gezeichnet:

Eine `while`-Schleife, die kontinuierlich Unterstreichungen zeichnet. Es wird `CGContextSetShadowWithColor` aufgerufen, um die Verschiebung, den Radius und die Farbe des Schattens festzulegen. Anschließend wird `YYTextDrawRun` aufgerufen, um das eigentliche Zeichnen durchzuführen. `YYTextDrawRun` wird an drei Stellen aufgerufen:

Es wird verwendet, um innere Schatten, Textschatten und Text zu zeichnen. Dies zeigt, dass es sich um eine universelle Methode handelt, um das Objekt `Run` zu zeichnen.

Zuerst wird die Transformationsmatrix des Textes abgerufen, um mit `runTextMatrixIsID` zu überprüfen, ob sie unverändert bleibt. Wenn es sich nicht um vertikales Layout handelt oder keine grafische Transformation festgelegt wurde, wird der Text direkt gezeichnet. Der Aufruf von `CTRunDraw` zeichnet das `run`-Objekt. Anschließend zeigt der Breakpoint, dass beim Zeichnen des anfänglichen Schattens nur der `if`-Block betreten wird, nicht der `else`-Block.

Also, damit ist unsere Schattenerstellung abgeschlossen!

Zusammenfassend speichert `YYLabel` zunächst Effekte wie Schatten in den Attributen des `attributedText`. Es überschreibt die `display`-Methode von `UIView` und führt darin asynchrones Zeichnen durch. Mit dem `CoreText`-Framework werden `CTLine`- und `CTRun`-Objekte erstellt, und aus `CTRun` werden die Attribute abgerufen. Anschließend wird basierend auf den Eigenschaften in den Attributen das `CTRun`-Objekt mit dem `CoreGraphics`-Framework in den Context gezeichnet.

Das Verständnis ist noch nicht ausreichend, ich werde später noch einmal darauf zurückkommen. Ich kann nicht umhin, zu bewundern, wie stark YY ist! Heute habe ich meine Gedanken sortiert und mich dazu gebracht, den Code beim Schreiben zu lesen, um es nicht zu langweilig zu machen, und gleichzeitig als Referenz für andere zu dienen. Ich muss jetzt schlafen gehen.