□□□□□□ कैसे काम करता है

ऊपर दिखाया गया छाया प्रभाव निम्नलिखित कोड का उपयोग करके बनाया गया है:

आप देख सकते हैं कि YYTextShadow बनाया गया है, फिर इसे attributedString के yy_textShadow को सौंपा गया है, और फिर attributedString को YYLabel में सौंपा गया है, और अंत में YYLabel को UIView में जोड़कर प्रदर्शित किया गया है। yy_textShadow को ट्रैक करने पर पता चलता है कि मुख्य रूप से textShadow को NSAttributedString के □□□□□□□□□□ से बांधा गया है, जहां □□□ YYTextShadowAttributeName है और □□□□□ textShadow है, यानी पहले □□□□□□ को स्टोर किया जाता है और बाद में उपयोग किया जाता है। □□□□□ + □□□□□□□ + □ का उपयोग करके परिभाषा पर तेजी से जाएं:

यहाँ एक addAttribute है, जो NSAttributedString.h में परिभाषित है:

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

यह कोड □□□□□□□□□□-□ में एक मेथड को दर्शाता है जो किसी विशेष गुण (□□□□□□□□□□) को एक निश्चित रेंज (□□□□□) में जोड़ने के लिए उपयोग किया जाता है। इसमें name पैरामीटर गुण का नाम, value पैरामीटर गुण का मान, और range पैरामीटर उस रेंज को दर्शाता है जहाँ यह गुण जोड़ा जाएगा।

यह बताता है कि इसमें किसी भी की-वैल्यू जोड़ी को असाइन किया जा सकता है। और YYTextShadowAttributeName की परिभाषा एक सामान्य स्ट्रिंग है, जिसका मतलब है कि पहले □□□□□□ की जानकारी को स्टोर किया जाता है, और फिर बाद में उपयोग किया जाता है। हम YYTextShadowAttributeName को ग्लोबली सर्च करते हैं।

फिर हम YYTextLayout में YYTextDrawShadow फ़ंक्शन पर आते हैं:

CGContextTranslateCTM का मतलब है कि यह एक □□□□□□□ के अंदर मूल बिंदु (□□□□□□) के निर्देशांक को बदलता है, इसलिए

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

(यह कोड □□□□□□□□□□-□ में है और इसे अनुवादित नहीं किया जाता है। यह कोड CGContextTranslateCTM फ़ंक्शन का उपयोग करके ग्राफ़िक्स कॉन्टेक्स्ट को ट्रांसलेट करता है।)

यह कहा जा रहा है कि ड्रॉ किए गए कॉन्टेक्स्ट को point बिंदु पर ले जाना है। हम पहले यह समझ लेते हैं कि YYTextDrawShadow कहाँ कॉल किया गया है, और हमने पाया कि यह drawInContext में कॉल किया गया है।

drawInContext में, निम्नलिखित क्रम में ब्लॉक की सीमा रेखा खींची जाती है: पहले बॉर्डर, फिर पृष्ठभूमि बॉर्डर, छाया, अंडरलाइन, टेक्स्ट, एक्सेसरीज़, इनर शैडो, स्ट्राइकथ्रू, टेक्स्ट बॉर्डर, और अंत में डिबग लाइन।

तो आखिर drawInContext का उपयोग कहाँ किया गया है? आप देख सकते हैं कि इसमें एक पैरामीटर YYTextDebugOption है, इसलिए यह फ़ंक्शन निश्चित रूप से सिस्टम का कॉलबैक नहीं है, बल्कि □□□□□□ के अंदर से ही कॉल किया गया है।

□□□□ + 1 दबाने पर शॉर्टकट पॉप अप होता है, और यह देखा जाता है कि इसे चार स्थानों पर कॉल किया गया है।

drawInContext:size:debug अभी भी □□□□□□ का अपना कॉल है, क्योंकि debug का प्रकार YYTextDebugOption * है, जो □□ का अपना है। newAsyncTask सिस्टम कॉल की तरह नहीं लगता, addAttachmentToView:layer: भी ऐसा ही है, इसलिए संभावना है कि यह drawRect: हो सकता है।

देखिए, दाईं ओर के त्वरित सहायता अनुभाग में विस्तृत व्याख्या है, और सहायता के नीचे यह भी बताया गया है कि यह UIView में परिभाषित है। अब YYTextContainerView को देखें, यह UIView को इनहेरिट करता है।

तो क्या YYLabel ने YYTextContainerView का उपयोग किया है? और फिर सिस्टम को YYTextContainerView में drawRect: को कॉल करके ड्रा करने दिया है?

अजीब बात है, YYLabel UIView को इनहेरिट करता है। इसलिए, □□□□□□ में दो चीज़ें होनी चाहिए! एक YYLabel और एक YYTextView, जैसे UILabel और UITextView। फिर हम पहले वाले YYLabel के newAsyncDisplayTask को फिर से देखेंगे।

बहुत लंबा है, बीच में YYTextLayout के drawInContext को कॉल किया गया है। newAsyncDisplayTask, यह कहाँ कॉल किया गया है?

दूसरी पंक्ति में कॉल किया गया है। इसलिए इसे सरलता से समझा जा सकता है कि YYLabel ने टेक्स्ट को ड्रॉ करने के लिए एसिंक्रोनस तरीके का उपयोग किया है। और _displayAsync को ऊपर के display द्वारा कॉल किया गया है। display के डॉक्यूमेंटेशन को देखने पर, यह कहा गया है कि सिस्टम उचित समय पर लेयर की सामग्री को अपडेट करने के लिए इसे कॉल करेगा, आपको इसे सीधे कॉल नहीं करना चाहिए। हम इसमें एक ब्रेकपॉइंट भी सेट कर सकते हैं।

display को CALayer के एक □□□□□□□□□□□ में कॉल किया जाता है। □□□□□□□□□□□ का उपयोग शायद इसलिए किया जाता है ताकि बड़े पैमाने पर अपडेट को एक साथ किया जा सके, जिससे □□□□□□□□□□ बढ़ सके। यह □□□□□□□□ में □□□□□□□□ की आवश्यकता की तरह नहीं लगता।

display □□□□□□□□□□□□□□□□□□□□ □□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

तो, हमारे पास थोड़ा सा विचार है। YYLabel ने UIView के display मेथड को ओवरराइड करके अपनी छाया और अन्य प्रभावों को एसिंक्रोनस तरीके से ड्रॉ करने का तरीका अपनाया है। छाया प्रभाव पहले YYLabel के attributedText में एट्रिब्यूट के रूप में सहेजे जाते हैं, और फिर display में ड्रॉ करते समय इन्हें निकाला जाता है। ड्रॉ करते समय सिस्टम के □□□□□□□□□□□□ फ्रेमवर्क का उपयोग किया जाता है।

कुछ विचारों को स्पष्ट करने के बाद, आपको पता चलेगा कि वास्तव में शक्तिशाली क्या है? एक तरफ इतने सारे प्रभावों और एसिंक्रोनस कॉल्स को व्यवस्थित करना है, और दूसरी तरफ □□□□□□□□□□□□ फ्रेमवर्क का गहराई से उपयोग करना है। इसलिए, पिछले कोड संगठन को समझने के बाद, हम □□□□□□□□□□□□ फ्रेमवर्क में गहराई से जाएंगे। देखें कि यह कैसे ड्रॉ किया जाता है।

चलिए फिर से YYTextDrawShadow पर वापस आते हैं।

यहां, CGContextSaveGState और CGContextRestoreGState ने ड्राइंग कोड के एक ब्लॉक को घेरा हुआ है। CGContextSaveGState का मतलब है कि वर्तमान ड्राइंग स्टेट की एक कॉपी बनाकर उसे ड्राइंग स्टैक में डाल दिया जाए। हर ड्राइंग □□□□□□□□ एक ड्राइंग स्टैक को मेन्टेन करता है। मुझे यह स्पष्ट नहीं है कि स्टैक के अंदर वास्तव में क्या होता है। फिलहाल इसे इस तरह समझें कि ड्राइंग □□□□□□□□ से पहले CGContextSaveGState को कॉल करना है और ड्राइंग □□□□□□□□ के बाद CGContextRestoreGState को कॉल करना है, ताकि बीच का ड्राइंग □□□□□□□□ में प्रभावी ढंग से दिखाई दे। CGContextTranslateCTM □□□□□□□□ को संबंधित स्थान पर ले जाता है। पहले point.x और point.y पर ले जाया जाता है, जो ड्राइंग का संबंधित स्थान है, लेकिन बाद में 0 और size.height पर ले जाने का कारण स्पष्ट नहीं है, इसे बाद में देखा जाएगा। फिर lines को निकाला जाता है और for लूप को एक्ज़ीक्यूट किया जाता है।

lines क्या है? यह YYTextLayout में (YYTextLayout *)layoutWithContainer:(YYTextContainer *)container text:(NSAttributedString *)text range:(NSRange)range के दौरान असाइन किया जाता

है।

फिर इस फ़ंक्शन की परिभाषा पर जाएं:

यह फ़ंक्शन बहुत लंबा है, 367 से 861 लाइन तक, 500 लाइन का कोड! इसके शुरुआत और अंत को देखकर, यह स्पष्ट है कि इसका उद्देश्य इन चरों को प्राप्त करना है। `lines` कैसे प्राप्त किया जाता है?

एक बड़े `for` लूप में, एक-एक करके `line` को `lines` में जोड़ा जा सकता है। तो `lineCount` कैसे प्राप्त होता है?

472वीं पंक्ति में एक `framesetter` ऑब्जेक्ट बनाया गया है, `text` पैरामीटर `NSAttributedString` है, फिर `frameSetter` ऑब्जेक्ट में एक `CTFrameRef` बनाया गया है, और फिर `CTFrameRef` से `lines` प्राप्त किया गया है। `line` वास्तव में क्या है? हम इस पर एक ब्रेकपॉइंट लगाएंगे।

पाया गया कि `shadow` शब्द का `lineCount = 2` है, जो हमारी कल्पना के अक्षरों की संख्या नहीं है।

तो क्या अनुमान लगाया जा सकता है कि सफेद `Shadow` पूरी तरह से एक `line` है, और छाया भी एक `line` है?

□□□□□□ में कई उदाहरण हैं, जिनमें से केवल एक प्रभाव दिखाया गया है, और अन्य कोड को टिप्पणी में डाल दिया गया है। यह देखकर अजीब लगता है कि □□□□□□ का `lineCount = 2` है, और □□□□□□□□ □□□□□□□ का `lineCount` भी 2 है, लेकिन □□□□□□□□ □□□□□□□ में आंतरिक छाया (□□□□□ □□□□□□) भी है, इसलिए यह 3 होना चाहिए?

□□□□□□ के □□□□□ डॉक्यूमेंटेशन को देखें, यह कहता है कि □□□□□□ एक पंक्ति के टेक्स्ट को प्रतिनिधित्व करता है, और एक □□□□□□ ऑब्जेक्ट में `glyph runs` का एक समूह होता है। तो यह सिर्फ साधारण पंक्तियों की संख्या है! ऊपर दिए गए ब्रेकपॉइंट स्क्रीनशॉट को देखें, जहां `shadow` का मान 2 था, क्योंकि इसका टेक्स्ट `shadow\n\n` था। ध्यान दें कि `\n\n` को जानबूझकर जोड़ा गया था, ताकि यह सुंदर दिखे।

इसलिए `shadow\n\n` दो पंक्तियों का टेक्स्ट है। □□□□□□ वही है जिसे हम आमतौर पर एक पंक्ति कहते हैं। फिर हमारे `lineCount` पर वापस जाते हैं:

यहां हमें `CTLines` सरणी मिलती है, उसमें से तत्वों की संख्या प्राप्त करते हैं, और यदि `lineCount` 0 से अधिक है तो प्रत्येक पंक्ति के निर्देशांक मूल बिंदु प्राप्त करते हैं। ठीक है, अब हमारे पास `lineCount` है, आइए अब `for` लूप पर नजर डालते हैं।

`ctLines` सरणी से `CTLine` प्राप्त करें, फिर `YYTextLine` ऑब्जेक्ट प्राप्त करें, और उसे `lines` सरणी में जोड़ें। फिर `line` के फ्रेम की गणना करें। `YYTextLine` का कंस्ट्रक्टर बहुत सरल है, पहले स्थिति, लंबवत टाइपसेटिंग, और `CTLine` ऑब्जेक्ट को सहेजता है:

`lines` को समझने के बाद, हम पहले वाले `YYTextDrawShadow` पर वापस जाते हैं:

यह कोड अब सरल हो गया है। पहले पंक्तियों की संख्या प्राप्त करें, उन्हें ट्रैवर्स करें, फिर `GlyphRuns` सरणी प्राप्त करें, और उसे ट्रैवर्स करें। `GlyphRun` को एक प्राइमिटिव या ड्रॉइंग यूनिट के रूप में समझा जा सकता है। फिर उससे `attributes` सरणी प्राप्त करें, हमारे पहले `YYTextShadowAttributeName` का उपयोग करके, हम शुरू में असाइन किए गए `shadow` को प्राप्त करें, और फिर छाया बनाना शुरू करें:

एक `while` लूप, जो लगातार उप-छाया बनाता है। `CGContextSetShadowWithColor` को कॉल करके छाया का विस्थापन, त्रिज्या और रंग सेट किया जाता है। फिर `YYTextDrawRun` को कॉल करके वास्तविक रूप से ड्रॉ किया जाता है। `YYTextDrawRun` को तीन स्थानों पर कॉल किया जाता है:

यह आंतरिक छाया, टेक्स्ट छाया और टेक्स्ट को ड्रॉ करने के लिए उपयोग किया जाता है। यह दर्शाता है कि यह एक सामान्य विधि है, जिसका उपयोग `Run` ऑब्जेक्ट को ड्रॉ करने के लिए किया जाता है।

शुरुआत में टेक्स्ट का ट्रांसफॉर्म मैट्रिक्स प्राप्त करें, और `runTextMatrixIsID` का उपयोग करके जांचें कि क्या यह अपरिवर्तित है। यदि यह वर्टिकल टाइपसेटिंग नहीं है या ग्राफिक्स ट्रांसफॉर्मेशन सेट नहीं है, तो सीधे ड्रॉ करें। `CTRunDraw` को कॉल करके `run` ऑब्जेक्ट को ड्रॉ करें। फिर ब्रेकपॉइंट पर ध्यान दें, और पाएं कि शुरुआती शैडो को ड्रॉ करते समय केवल `if` ब्लॉक में प्रवेश किया गया था, `else` ब्लॉक में नहीं।

तो हमारी छाया ड्राइंग यहीं समाप्त होती है!

संक्षेप में, `YYLabel` पहले छाया जैसे प्रभावों को `attributedText` के □□□□□□□□□□ में सहेजता है, `UIView` की `display` विधि को ओवरराइड करता है, और `display` में एसिंक्रोनस ड्राइंग करता है। `CoreText` फ्रेमवर्क का उपयोग करके `CTLine` और `CTRun` ऑब्जेक्ट प्राप्त करता है, `CTRun` से □□□□□□□□□□ प्राप्त करता है, और फिर □□□□□□□□□□ में मौजूद विभिन्न गुणों के आधार पर `CoreGraphics` फ्रेमवर्क का उपयोग करके `CTRun` ऑब्जेक्ट को □□□□□□□ में ड्रा करता है।

समझ अभी भी पर्याप्त नहीं है, बाद में फिर से पढ़ने के लिए आऊंगा। अनजाने में यह सोचकर आश्चर्य होता है कि □□ वास्तव में बहुत शक्तिशाली है! आज मैंने अपने विचारों को व्यवस्थित किया, और खुद को कोड लिखते और पढ़ते हुए रखा, ताकि यह उबाऊ न हो, और साथ ही दूसरों के लिए संदर्भ के रूप में काम कर सके। अब सोने जाना होगा।