How YYText Works

The shadow effect above is achieved using the following code:

First, a `YYTextShadow` object is created and assigned to the `yy_textShadow` attribute of the `attributedString`, which is then assigned to a `YYLabel`. The `YYLabel` is added to a `UIView` for display. By tracing `yy_textShadow`, we find that it is stored in the `NSAttributedString` attributes with the key `YYTextShadowAttributeName`. This means the shadow is stored for later use. Using Shift + Command + J, we can quickly jump to the definition:

Here, `addAttribute` is defined in `NSAttributedString.h`:

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

This means you can assign any key-value pair to it. The definition of `YYTextShadowAttributeName` is just a regular string, indicating that the shadow information is stored for later use. Let' s search for `YYTextShadowAttributeName` globally.

Now, let' s move to the `YYTextLayout` function `YYTextDrawShadow`:

`CGContextTranslateCTM` changes the origin coordinates in a Context, so

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

means moving the drawing context to the point specified. First, we need to understand where `YYTextDrawShadow` is called, which we find in `drawInContext`.

In `drawInContext`, various elements are drawn in sequence, including the block' s border, background border, shadow, underline, text, attachments, inner shadow, strikethrough, text border, and debug lines.

So where is `drawInContext` used? Given the `YYTextDebugOption` parameter, this function is likely not a system callback but called within YYText itself.

Using Ctrl + 1, we find that it is called in four places.

`drawInContext:size:debug` is likely still within YYText, as `debug` is of type `YYTextDebugOption *`. `newAsyncTask` and `addAttachmentToView:layer:` also seem to be internal calls, making it likely that `drawRect:` is involved.

Indeed, `drawRect:` is defined in `UIView`. `YYTextContainerView` inherits from `UIView`.

So, `YYLabel` likely uses `YYTextContainerView` to let the system call its `drawRect:` method for rendering.

Interestingly, `YYLabel` itself inherits from `UIView`. Hence, YYText might have two sets of components: `YYLabel` and `YYTextView`, similar to `UILabel` and `UITextView`. Returning to the `newAsyncDisplayTask` in `YYLabel`, in the middle, it calls `drawInContext` from `YYTextLayout`. The `newAsyncDisplayTask` itself is called in `_displayAsync`.

In `_displayAsync`, it is called on the second line. So, `YYLabel` uses asynchronous rendering for text. `_displayAsync` is called by `display`, which is mentioned in the documentation to be called by the system at appropriate times to update the layer' s content. Let' s set a breakpoint on `display`.

This shows `display` is called within a transaction in `CALayer`. Transactions are likely used for batch updates for efficiency, rather than for rollback purposes as in databases.

The documentation also mentions that if you want your layer to render differently, you can override this method to implement custom rendering.

Thus, we have a rough idea: `YYLabel` overrides `UIView`' s `display` method for asynchronous rendering of shadows and other effects. The shadow effect is first stored in the `attributedText`' s attributes and retrieved during rendering using the CoreGraphics framework.

Understanding the code organization helps us realize the strength lies in managing various effects and asynchronous calls, along with proficient use of the CoreGraphics framework. With this understanding, let' s delve deeper into the CoreGraphics framework to see how the rendering is done.

Returning to `YYTextDrawShadow`,

`CGContextSaveGState` and `CGContextRestoreGState` surround a segment of drawing code. `CGContextSaveGState` copies the current drawing state onto a stack, maintained by each drawing context. Without knowing the exact internal operations of this stack, we can understand that `CGContextSaveGState` should be called before and `CGContextRestoreGState` after drawing to ensure effective rendering within the context. `CGContextTranslateCTM` moves the context to the specified position. Moving to `point.x` and `point.y` makes sense, but the subsequent move to 0 and `size.height` requires further investigation. The `lines` are then obtained and iterated over.

`lines` is assigned in `layoutWithContainer:text:range:` in `YYTextLayout`.

This function is quite lengthy, from lines 367 to 861, totaling 500 lines. Its purpose is to obtain these variables. How are `lines` obtained?

Within a large `for` loop, individual lines are added to `lines`. The `lineCount` is then determined.

At line 472, a `framesetter` object is created with `text` as an `NSAttributedString`, and a `CTFrameRef` is obtained from it, which in turn provides the `lines`. Let' s set a breakpoint to understand what a `line` is.

Interestingly, `lineCount` for the word `shadow` is 2, not the number of letters.

This suggests that the entire word `Shadow` is one line, and the shadow itself is another line.

Looking at examples in YYText, only one effect is shown while commenting out the others. It’ s odd that `lineCount` is 2 for both `Shadow` and `Multiple Shadows`, even though `Multiple Shadows` should have three lines with an inner shadow.

The Apple documentation for CTLine indicates that a CTLine represents a single line of text, containing a group of glyph runs. So, it essentially represents the number of lines. The previous breakpoint screenshot shows `shadow` as `shadow\n\n`, where `\n\n` was added intentionally for display purposes.

So `shadow\n\n` is two lines of text. CTLine is simply a line. Returning to `lineCount`:

Here, the `CTLines` array is obtained, and its count is used to determine `lineCount`. If `lineCount` is greater than 0, the origin coordinates of each line are obtained. With `lineCount` determined, let’ s proceed to the `for` loop.

From the `ctLines` array, a `CTLine` is obtained, which is then converted into a `YYTextLine` object and added to `lines`. Some frame calculations are performed for each line. The `YYTextLine` constructor simply stores the position, vertical layout status, and `CTLine` object.

Having understood `lines`, let’ s return to `YYTextDrawShadow`:

The code now becomes straightforward. The number of lines is obtained and iterated over, followed by extracting the `GlyphRuns` array for each line, which represents drawing units or primitives. From each `GlyphRun`, the attributes array is obtained, and the previously assigned `shadow` is retrieved using `YYTextShadowAttributeName`. Shadow drawing begins:

A `while` loop continuously draws sub-shadows. `CGContextSetShadowWithColor` sets the shadow offset, radius, and color. `YYTextDrawRun` is then called to actually draw. `YYTextDrawRun` is called in three places:

It’ s used for drawing inner shadows, text shadows, and text itself, indicating it’ s a general-purpose method for drawing `Run` objects.

Initially, the text transformation matrix is obtained, and if the run is not vertically laid out or lacks a transformation, it is drawn directly using `CTRunDraw`. The breakpoint reveals that while drawing the initial shadow, only the `if` block is entered, not the `else` block.

Thus, shadow rendering ends here!

In summary, `YYLabel` stores shadow effects in the `attributedText` attributes, overrides the `UIView`’ s `display` method for asynchronous rendering, retrieves the attributes during rendering, and uses the CoreGraphics framework to draw the `CTRun` objects onto the context.

Further understanding is needed, which can be gained through future study. YY is truly impressive! Today, I organized my thoughts, making it easier to read and write code simultaneously, hopefully serving as a reference for others. Time for bed.