TextSizingExample

Description

This example demonstrates different ways of configuring the objects in the new text system that comes with OPENSTEP for Windows and Mach.   The app displays a window where different examples of NSTextViews with different sizing properties can be shown.   A pop-up button at the top of the window controls which example is visible.

In most of the examples, the same NSTextStorage is used and the example contains an NSLayoutManager that adds itself to the shared NSTextStorage and one or more pairs of NSTextContainer and NSTextView objects.   By default this NSTextStorage loads this ReadMe file, but you can load a new file or edit the text.

Implementation

The Controller class acts as the NSApplication delegate and provides a very simple swap-view type functionality.   The Controller also provides the shared NSTextStorage.   It has an array of Aspect objects that it lists in the pop-up and switches the main part of the window to show the chosen example.

Each example is controlled by a specific subclass of the Aspect object. Aspect itself provides the basic swapping support and a nib file that is sufficient for most of the examples.   The subclasses of Aspect contain almost all the code relevant to the example.   Each one implements the method -nibDidLoad.   This method creates and configures all the objects involved in the particular example.   In general, the code is more explicit than it needs to be.   Many features of the objects are explicitly set to their default values to make the examples as clear as possible.

This example defines a number to use to mean "really big".   It is called LargeNumberForText, and it was not arbitrarily chosen.   The actual value was chosen to be around the largest floating point value possible that can preserve at least pixel precision.   Because of the nature of floating point numbers, the bigger they get, the less precise they get.   It is not wise to use bigger dimensions for text system objects because, even if you ever fill all that space, by the time you get to the far reaches, the letters won't have the precision necessary to look and act correctly.   This limitation of floating point coordinates goes all the way down into postscript, and holds for any type postscript graphics.

Text Sizing Configuration

This document assumes at least a passing familiarity with the new text system.   Also, for lack of a better term, we will sometimes use the singular "text object" to refer to a fully functional stack of text objects including an NSTextStorage, NSLayoutManager, and one or more pairs of NSTextContainer and NSTextView objects.

Because there are so many different ways in which text is used in an application, the text system has quite a few options for configuring the sizing behavior of a particular text object.   This can be confusing to developers new to the system, but it also enables you to configure a text object for almost sizing needs once you understand the options.

Both the NSTextView and NSTextContainer classes have configurable sizing options. In order to get exactly the behavior you desire, it is usually necessary to set up the options in both these objects. A list of the API for these options follows. Only the set methods are mentioned, but access methods exist for each setting as well. Also included is some relevant API inherited by NSTextView from NSText and NSView.

NSTextContainer

-€(void)**setContainerSize:**(NSSize)*aSize*
-€(void)**setWidthTracksTextView:**(BOOL)*flag*
-€(void)**setHeightTracksTextView:**(BOOL)*flag*

NSTextView

-€(void)**setFrame:**(NSRect)*frameRect*
-€(void)**setFrameOrigin:**(NSPoint)*newOrigin*
-€(void)**setFrameSize:**(NSSize)*newSize*
-€(void)**setConstrainedFrameSize:**(NSSize)*desiredSize*
-€(void)**setAutoresizesSubviews:**(BOOL)*flag*
-€(void)**setAutoresizingMask:**(unsigned€int)*mask*
-€(void)**setHorizontallyResizable:**(BOOL)*flag*
-€(void)**setVerticallyResizable:**(BOOL)*flag*
-€(void)**setMinSize:**(NSSize)*newMinSize*
-€(void)**setMaxSize:**(NSSize)*newMaxSize*

The first concept to understand is that it is the size of the NSTextContainer, not the NSTextView which controls the area where the text will be laid out. The size of an NSTextContainer does not change because of the layout of the text, it determines the layout of the text. Usually the size of a container does not change at all. However the two methods

-setWidthTracksTextView: and **-setHeightTracksTextView:** can be used to enable a special kind of dynamic sizing of the container. You can set it to change its width or height as its NSTextView changes.

The NSTextView on the other hand changes size much more often. An NSTextView will automatically size itself to match the usage of the container. The usage of a container is effectively how much of it is actually filled up with text (see the NSLayoutManager documentation for a discussion of text container usage). NSTextView can have constraints placed upon it in to limit it size changing. You can disable or enable the sizing to usage in a particular dimension using **-setHorizontallyResizable:**, and **-setVerticallyResizable:**. Alternatively you can constrain the size with minimum and maximum boundaries using **-setMinSize:**, and **-setMaxSize:**. Note that disabling horizontal resizing effectively causes the widths of the min and max sizes to be ignored (and the same goes for vertical resizability and the min and max heights).

Warning: It is possible to set up an inconsistent state between an NSTextContainer and an NSTextView. If you turn on both width tracking in the container and horizontal sizability in the text view at the same time, you can end up causing infinite loops in your application. Consider the configuration: suppose the usage of the container changes and the text view's width changes in response. Then the text container will notice this and change its width to match. The change in the text container's size will cause the text to be relaid out to the new container width. This may cause the container's usage to change again. Again the text view will change its width, and the loop is complete. Avoid this configuration and the analogous one with height tracking and vertical resizability.

Another thing to keep in mind is that only the automatic resizing of the text view to match the container usage is constrained by the min and max sizes and the sizability settings of the text view. **-setFrameSize:** does not apply the constraints. The constraints are applied only through the **-setConstrainedFrameSize:** method. Among other things, this means that the NSView autosizing (springs and rods) feature does not respect the constraints.

The rest of this file discusses the implementation of each example.

VertScrollerAspect

This demonstrates the simple case of an NSScrollView with a vertical scroller containing an NSTextView that grows vertically to fit the text and wraps the text to its width, rewrapping if the width changes. This example creates NSLayoutManager, NSTextContainer, and NSTextView objects and hooks the NSLayoutManager into the shared NSTextStorage for the app.

NSTextContainer
Size: (view width, LargeNumberForText)
Tracks width: YES
Tracks height: NO

NSTextView
Size: (content width of scroll view, container usage height)
Horizontally resizable: NO
Vertically resizable: YES

Min Size: (irrelevant, content height of scroll view)
Max Size: (irrelevant, LargeNumberForText)

Notice that the container tracking and the text view sizability along a single dimension are different. This should always be the case (see the warning in the Text Sizing Configuration section above).

The container height does not change. It is set to be really tall so that, effectively, we can just keep laying text out down the container and never run out of room. The width, on the other hand, is set to change as the width of the text view changes. The text view is never allowed to get smaller than the content rect of the scroll view (this is actually taken care of automatically for NSTextViews that are the documentView of a NSClipView), but it is allowed to get as tall as it wants. It does not change its width in response to usage changes for its text container at all.

The autosizing flags of the text view are set so that it will resize its width with its superview. This setup causes the text to be wrapped to the width of the text view, but to grow downward as needed. If the user resizes the window wider, the text view will change width and the text will be rewrapped to the new width. This is the classic Edit style of text setup.

BiScrollerAspect

This demonstrates non-wrapping text in an NSScrollView. The scroll view allows scrolling in both directions now, and the text does not wrap at all except at hard line breaks. This is the mode that people familiar with Windows code editors are probably used to.

NSTextContainer

Size: (LargeNumberForText, LargeNumberForText)
Tracks width: NO
Tracks height: NO

NSTextView

Size: (container usage width, container usage height)
Horizontally resizable: YES
Vertically resizable: YES
Min Size: (content width of scroll view, content height of scroll view)
Max Size: (LargeNumberForText, LargeNumberForText)

The container never changes size in this example. It is huge in both dimensions to give us plenty of horizontal and vertical room. The view will size itself to the container usage in both directions. It will be as wide as the widest line and tall enough to display all the lines. No autosizing is used for the text view since it should always stay sized to the container usage.

FixedSizeAspect and TwoColumnsAspect

In a way the FixedSizeAspect and the TwoColumnsAspect are both the simplest and most complex configurations. They are simple in their sizing behavior. Basically, they are fixed size. They do not resize at all to their containers' usages, although they do have autosizing attributes for when the window resizes. They are the most complex in that the situations in which these setups are useful are usually more advanced uses of the text system such as where the application is doing its own pagination.

These two Aspects are very similar. They have almost identical configurations of their text views and containers. The difference is that TwoColumnsAspect has two NSTextContainer/NSTextView pairs instead of just one.

NSTextContainer
Size: (view width, view height)
Tracks width: YES
Tracks height: YES

NSTextView
Size: determined by window size
Horizontally resizable: NO
Vertically resizable: NO
Min Size: irrelevant
Max Size: irrelevant

The size of the NSTextView and also the NSTextContainer is determined solely by the window size. When the text view resizes, the container resizes to match it in both dimensions. The text view never changes size in response to the container usage. When the container(s) fills up, the rest of the text goes nowhere.

In the FixedColumnAspect example the single view autosizes in both directions to fill the available space in the window. In the TwoColumnsAspect, both columns autoresize their height, but only the right column autosizes its width.

FieldAspect

This aspect is different from the others in that it does not use the shared NSTextStorage from the Controller. Instead it demonstrates how to set up the text system to act in a field-like way with various different sizing behaviors. It has six different text objects. The first three are growing fields of various alignments and the second group of three are fixed size areas, but horizontally scrolling.

In the first three examples the NSTextViews resize, taking only as much space as necessary, but never more than they have available. The centered and right aligned variants are handled specially so that as the views grow or shrink to fit the text, they are kept centered or right aligned within their boxes. The left aligned field requires no special concern for this. The important thing to remember is that NSTextView may resize itself to fit its text, but it never moves itself. For the centered and right aligned examples to work correctly, the views must move sometimes too, and we have to handle that ourselves. These examples can display only as much text as will fit in the current width of the boxes that they are in.

In the second three examples, the NSTextViews still resize to fit their text, but they are kept at least as large as the width of the box they are in. Furthermore, the NSTextViews are in NSClipViews so that if they get wider than the box the user can scroll them like a scrolling NSTextField. No special code is required for the various different alignments here.

This example illustrates another feature of the text system unrelated to resizing. The NSTextViews in the FieldAspect are set to be field editors. This means that they will send textDidEndEditing: notifications when the user types tab, shift tab, or return while editing in one. This is the full extent to which NSTextView participates in the nextKeyView mechanism (keyboard

UI). NSTextView does not automatcally select its next key view. The most it does is send NSTextDidEndEditingNotifications in this way. It is usually the delegate of an NSTextView that actually handles this notification and causes a new view to be made key, if appropriate. NSTextField and NSMatrix, for example, use an NSTextView for editing editable text. They handle these notifications to pass on the key view status to their own nextKeyView or previousKeyView (or in the case of NSMatrix, perhaps the next or previous cell). The FieldAspect also handles these notifications. It takes care of distinguishing why the editing ended and takes the appropriate action. This is why you are able to tab through the fields in the aspect.