slimPicker: A Slimmer listPicker Proto

by Jeremy Wyld and Maurice Sharp, Apple Computer, Inc.

The Newton 2.0 OS provides many new prototypes for developers to use. One of the popular ones is `protoListPicker`. It was designed to provide a generalized framework and interface for presenting lists of choices to the user. Unfortunately, `protoListPicker` also tries to do everything for the developer. The result is a complex API, and overhead in space and time that is not necessary in a lot of cases. This article presents a slimmer picker.

The Data Is the Picker (listPicker)

Most of the overhead from listPicker is due to the way that it handles data. The purpose of listPicker is to display lists of data that the user can select items from. The data items can be elements in an array, soup entries, or a mix of both. To make the developers' life easier, listPicker requires some understanding of what the data is and how it is formatted. This is where the pickerDef and nameRef structures come from.

A nameRef is a generic data wrapper. It can be used to wrap an array element or a soup entry. All of the relevant data slots are part of the top level frame of the nameRef so that listPicker does not need to modify the actual data referenced by the nameRef. The pickerDef is the code object responsible for creating and managing nameRefs.

The pickerDef and nameRef structure provides flexibility, but data that is only array based or only soup based pays the overhead for representing the mixed array/soup data. Every line of data displayed in the listPicker requires a nameRef structure which requires heap space. Each nameRef created requires several levels of function calls to the pickerDef object. Each access to a nameRef requires several levels of functions calls. This is true even for a simple access such as comparison. Caching the data in the nameRef can sometimes avoid the calling overhead, but it costs heap space.

The pickerDef/nameRef abstraction does provide some benefits. In addition to the obvious mixing of array and soup items, it also enables listPicker to render the individual data display lines with little developer intervention. It also makes filing easy. On the downside, the representation is quite brittle. The seemingly simple task of using an icon as the first item in the rendered line of data is actually very difficult to implement.

A hidden cost is the complexity of learning to use listPicker. To create a simple picker that displays developer data requires understanding protoListPicker, the pickerDef object (such as protoNameRefDataDef) and the nameRef wrapper. It also requires learning how these three entities interact. There is no clear delineation of data manipulation and display characteristics. A good example of this is single selection, which is a characteristic of the pickerDef not the listPicker.

Another hidden cost is the “cursor” used by the listPicker. In order to handle both array- and soup-based data, the listPicker must implement a pseudo-cursor that can wrap both types of data. Unfortunately, the details of the implementation are hidden. This means that it is very difficult to use listPicker on data that is represented across multiple soups.

If you are displaying lists of names, or if your data can be both array and soup based, listPicker provides a nice proto for you to use. However, for most developers, all they want is to display soup-based data in a list. Enter slimPicker...

You Data, Me Picker (slimPicker)

When we set out to design slimPicker, we set four goals (well five, see below):

1 Provide the listPicker look and feel.

2 Minimize space and time costs.

3 Make it easy for the developer to understand and use.

4 Allow the developer to customize it with minimal effort.

Originally, we had a fifth goal of keeping the same API as listPicker. We hoped that a developer could just substitute slimPicker for listPicker and everything would work. However, we dropped that requirement after finding that supporting the API required very similar overhead to listPicker. Most of the overhead came from an iterator that could work with both soup cursor and arrays.

Once we dropped the requirement for supporting the API, we also dropped the pickerDef and nameRef structures. This contributes to all four goals. Most of the overhead of listPicker is in the pickerDef/nameRef call chains, as is most of the complexity and brittleness of implementation.

The rest of this section gives some examples of how the goals effected the design and implementation.

Look and Feel

The look and feel was easy to do. We examined listPicker to see which individual elements were used in its construction. We had the source code, but you could do a similar thing using DV in the inspector. We used the same elements in slimPicker with only one notable exception: instead of using a `protoStaticText` for the selection counter, we draw it. This saves a bit of time and space (though not much.)

The main part of slimPicker is implemented using a protoOverview because it can be used with either array- or soup-based data. It requires a cursor (or cursor-like) object for iterating over the data. The other nice feature of protoOverview is that it handles the selection check boxes.

The downside is that there is no supported way to cache the shapes used for each data display line. This results in a time penalty every time the display list of data lines is rebuilt. Unfortunately, this occurs any time you update the visual display list (that is, scrolling and the RefreshPicker call.) Luckily, slimPicker does this faster than listPicker. If slimPicker were built into ROM, we could overcome this difficulty by using undocumented features. However, we did not use these features because they are likely to change in future Newton OS devices.

Space and Time

The main performance gains come from making the developers responsible for their data. There is no pickerDef or nameRef structure to provide a generic data wrapper. Instead there is a well defined interface between slimPicker and the data. The developers are responsible for rendering the shape that is a line of data. They are also responsible for providing the cursor structure used by protoOverview, handling verification, creating new items (including any editing slips) and maintaining a list of the current selections.

Note that eliminating the pickerDef also eliminates the popup and validation overhead. In listPicker, the only way to determine if a given item in a column requires a popup character is to build the popup. That means each line of data built by listPicker requires a call to MakePopup in the pickerDef. In slimPicker, if a popup is required, the developer renders the popchar into the display line for that data. They also need to detect if a hit to that line is in the popable item, pop the correct picker and handle the result.

Although it seems like we have added more work for the developer, it is actually easier to implement a simple soup-based slimPicker than an equivalent listPicker.

In essence, we speeded up slimPicker and reduced the space by removing the data abstraction layer. This does increase the work a developer must do, but it also removes most of the overhead as well as reducing the complexity of the proto.

Easy to Understand and Use

In addition to eliminating the pickerDef/nameRef/listPicker interactions, we also reduced the overall number of slots and methods that a developer needs to learn. As an example, all of the listPicker "suppress" settings are now in one bit field called `visibleChildrenFlags`.

Another good example is adding new items. In listPicker you had to enable the New button and then provide several methods in your pickerDef. In addition, the callbacks for adding the new item are sent to the pickerDef context, not to the listPicker. In slimPicker, you enable the New button and provide one method called `CreateNewItem`. On the downside, slimPicker will not do any work such as bringing up a slip or calling back when the slip is dismissed. Instead, `CreateNewItem` is called when the New button is pushed. Everything else is up to you.

The change that provides the most flexibility is letting you render a line of data. You provide an `Abstract` method that is given the data item and a bounding box and returns a shape that represents the data item. That means you can put anything in that shape that is required, including icons and columns. In listPicker, adding an icon was difficult at best. In slimPicker, just add it to the shape for your data line.

slimPicker also provides an `AlphaCharacter` call that lets you return the character used for sorting a particular item of data. In listPicker you would have to provide at least one method (possibly two) and set up the column description. If the first displayed column of data was not the one used for sorting, there are additional methods and slots that must be provided. This means things like displaying icons in the first column is very difficult. For slimPicker, you can display what you wish, and control the order with `AlphaCharacter`.

A downside of slimPicker is that the developer is responsible for tracking selection. The developer needs to provide the `IsSelected` and `SelectItem` methods that do the right thing. Each change in selected state will require an update in the visual display of the data lines, which means redoing the children of the overview (that is, a call to `RefreshPicker`).

Single select is relatively easy to implement, use a single slot to represent the selected item and refresh the picker. Your `SelectItem` method will replace the value of the slot and your

`IsSelected` method will only return true if the entry passed in matches the one in the slot. The rest is handled by slimPicker.

Easy to Customize

Even though single selection is provided by listPicker (through the pickerDef!), it provides a nice example of how slimPicker is easy to customize. It also points out that, once again, eliminating the pickerDef/nameRef representation was a good decision.

Perhaps the biggest area of customization in slimPicker is the ability to change the data types and representation on the fly. In listPicker it is not possible to change the pickerDef or modify the cursor that access the data. The only way to do that is to close and open the listPicker. With slimPicker you have complete control over how the data is accessed (`cursor`) and how it is represented (`Abstract`). All that is required is a call to `RefreshPicker`, and everything will update.

Another example is filing. To add filing you just add the support you normally would in an application. That is, activate the folder tab (set the appropriate flag in `visibleChildrenFlags`), then provide the standard system API for filing (`appAll`, `NewFilingFilter`, etc.). When the filter changes, you can change your cursor and call `RefreshPicker`.

Six of One, a Dozen of Another

This section gives you some comparisons between listPicker and slimPicker. To be candid, the tests are stacked in favor of slimPicker. They are based on simple soup-based viewing.

All the tests were performed on the same MessagePad. MP 120, running Newton OS 2.0 and having 79 entries in Names. The listPicker-based FAX picker was opened from faxing a note and choosing "Other Names" from the Name picker. Heap space was measured before and after the picker was opened using HeapShow in the accurate setting with no timed updates. Timing started when the picker including "Other Names" was closed and stopped when the FAX picker was opened.

For the listPicker based People picker, we used the PeoplePicker-1 sample from Newton Developer Technical Support. Heap usage was measured using HeapShow. Timing started after the pen was released on the icon in the extras drawer and ended when the people picker appeared and was ready for input.

The slimPicker measurements were done on the protoSlimFaxPicker-1 and protoSlimPeoplePicker-1 code samples. These will be on Newton Developer CD #10 and on the web at http://dev.info.apple.com/newton/techinfo/slimPicker.html.

		listPicker	slimPicker
Heap Used in bytes	FAX picker	9300	3856
	People picker	8496	2740
Time to Open in seconds	FAX picker	5	3
	People picker	3	2

As you can see from the table, slimPicker is more efficient than listPicker in all cases. For heap usage this is not really surprising: listPicker has the overhead of nameRefs, a more complex selection tracking, extra cursors and a pickerDef. And slimPicker also has minimal extra information.

Time to open is a bit different. Both listPicker and slimPicker use protoOverview, but listPicker also has to construct the soup/array iterator and the nameRefs. In addition, each data access has the overhead of calling through the pickerDef object. slimPicker can just iterate over the visible data and call `Abstract` on each entry. There is very little housekeeping overhead.

Unfortunately, slimPicker is still fairly slow to launch. A quick profile of slimPicker shows that about 10% of the opening time is spent in the `Abstract` method of protoOverview. The other major piece is probably soup access. This means there is no effective way to speed up the code. The usual idea of caching is not useful since the slowness is part of the opening process. If the slowness was in scrolling, caching might makes sense, but of course that would increase the memory footprint.

One measure that is harder to estimate is the size of the object. We can get a good measure of slimPicker by either looking at the size of the package on the desktop, or by using TrueSize on the MessagePad. There is no similar way to find the size of listPicker.

API

This section presents the API to slimPicker. Of course, you will have all of the source code so you could modify anything. But just in case this code shows up in ROM someday, stick to the API.

Slots

`cursor`

This slot is required.

The iterator for the data displayed by the slimPicker. Can be either a soup cursor or a developer-defined object that implements the methods required by the cursor slot of protoOverview. See the Newton Programmers Guide or Newton Developer Technical Support Q&A for more information on the protoOverview cursor structure.

`folderTabTitle`

The text to put into the protoFolderTab. This is used to identify the slimPicker that is open. The default is NIL (i.e., no title).

You can use SetValue to change the value at runtime.

`reviewSelections`

If true, the slimPicker will only display the selected items. If NIL, all items will be displayed. Corresponds to the "Selected Only" checkbox in the user interface of the slimPicker. The default is NIL.

You can use SetValue to change the value at runtime.

`viewLineSpacing`

An integer representing the height of each line of data in the slimPicker. This value must be at least the height of the checkbox. The default is 14.

`visibleChildrenFlags`

Bit flags identifying which child views are to be visible. The values are:

Constant	**Value**	**Shows/Hides**
`vNewButton`	(1 << 0)	New button for adding new data items
`vScrollers`	(1 << 1)	Scrollers for scrolling the list
`vAZTabs`	(1 << 2)	AZTabs for alphabetical navigation
`vFolderTab`	(1 << 3)	Folder tab for filing
`vSelectionOnly`	(1 << 4)	Selection Only checkbox
`vCloseBox`	(1 << 5)	Close box for closing the slimPicker
`vCounter`	(1 << 6)	Count of selected items

The default is all views visible.

Methods

`slimPicker:Abstract(entry, bounds)`

This method is required.

Returns the shape that represents the given entry in the slimPicker. The shape must not be larger than bounds. This is where the developer renders an individual line of data. The returned shape must be one that DrawShape can use.

`slimPicker:AlphaCharacter(entry)`

This method is required if the AZTabs are visible.

Returns a character representing the index value for the given entry. The character will be used to set the appropriate tab in the AZTabs.

`slimPicker:CreateNewItem()`

This method is required if the New button is visible.

The method is called when the user taps the "New" button. The developer is responsible for any work that needs to be done to add the new entry. This includes creating and opening any editing or data entry slip, adding the data to the cursor, selecting the new item, and refreshing the slimPicker (see RefreshPicker below).

`slimPicker:GetHiliteShape(xcoord, bounds)`

This method is called to get the hilite box for a list item. The developer should provide this method if they wish to create a multiple column picker. This method should return something suitable for DrawShape.

`xcoord` is the current x coordinate of the pen normalized to bounds.

`bounds` is the bounding box for the item being hilited.

`slimPicker:GetNumSelected()`

This method is required if the counter is visible or `UpdateSelectedText` is called.

Returns the number of selected items.

`slimPicker:HitListItem(index, xcoord, ycoord)`

Called if a user has tapped on one of the items in the list. This method is called after the user has lifted the pen, not during the tracking of the hilite. It is only called if the tap occurred in the developer portion of the line. It is not called if the tap occurred in the checkbox.

The `index` is 0 based from the top of the displayed items. Note that cursor:Entry() corresponds to index 0. You can find the hit item by using cursor:Move(index).

`xcoord` and `ycoord` are the pen coordinates of the pen just before it was lifted. If you are displaying multiple columns, you can use these values to determine which column was hit. These coordinates are normalized to the picker list.

Note, the context of this call is the picklist embedded in the slimPicker proto.

The default method does nothing.

`slimPicker:IsSelected(entry)`

This method is required

Return true if the given entry is selected, NIL if it is not. Note that tracking the selected items is up to the developer. slimPicker provides no mechanism to do this.

`slimPicker:PickLetterScript(letter)`

This method is required if the AZTabs are displayed.

Called by the AZTabs in the slimPicker when the user selects a given tab. `letter` is a string containing a single character indicating which letter was tapped in the AZTabs.

This method should move the cursor to the appropriate entry and call RefreshPicker to update the displayed data.

`slimPicker:RefreshPicker()`

This method forces an update of the items in the slimPicker. This includes the picker itself, the AZTabs, and the arrows.

The data items will be re displayed from the current cursor entry.

`slimPicker:SelectItem(index)`

This method is required.

The user has clicked on the checkbox of the index'th data item. The developer should update their list of selected items accordingly. Note that the implementation details of keeping track of the selected items is up to the developer.

See HitItem above for a discussion of how to find the index'th data item.

`slimPicker:ToggleShowSelections(isOn)`

This method is required if the Selections Only checkbox is shown.

Called when the user taps on the "Selections Only" checkbox. `isOn` is true if the checkbox is checked, NIL if it is clear.

The developer should update the cursor to show either only selected items (`isOn` is true), or all items (`isOn` is nil). slimPicker will refresh based on this new cursor.

`slimPicker:UpdateSelectedText()`

Force the slimPicker to update the text that displays the number of selected items. Call this if you programatically change the selected items without user intervention.

Other Pieces

If you use the filing tabs, you must supply the slots and methods required by the standard protoFolderTab. See the Newton Programmers Guide information on filing for more information on these slots and methods.

If you are using soup-based data it is recommended that you register for soup change notification while the picker is open. That way you can programmatically update the picker using RefreshPicker when stores change or the soup is changed.