

# Creating ActionScript 3.0 components in Flash CS3 Professional – Part 8: Keyboard support

Jeff Kameron

Adobe

Welcome to Part 8 of the article series on creating components using ActionScript 3.0. This part will continue on from the previous segment, where we discovered how to control the focus of the elements within the MenuBar component. In this article we'll cover how to add keyboard support so that users can negotiate through the menu using key commands, as well as their mouse. If you didn't see the articles leading up to this section of the series, you may find it beneficial to go back and review the previous articles before embarking on this one. In Part 1 of the series you can download the sample files for the entire series. Or if you'd like to just download the sample files and follow along with this part of the series, you'll find the link below.

Whenever you implement focus management for your component, you should always add keyboard support as well. When a user tabs to a component, they will expect to be able to continue using the keyboard to select items within it. For the purposes of our sample project, I added keyboard support to the MenuBar component. I updated the code to allow the arrow keys to navigate the menus, the escape key to close the menus and also added functionality so that the space bar and enter key can be used to select a menu item. The details of keyboard support for the MenuBar component are sufficiently complicated that I created a side bar titled MenuBar keyboard support to provide the full explanation of how this works.

## Handling keyDown and keyUp Events

If your component needs to handle either the `keyDown` or `keyUp` event, you do not need to add your own listener because `UIComponent` registers `keyDownHandler()` and `keyUpHandler()` and you can override these methods. The `UIComponent` implementations do nothing, so you do not always need to call the `super` implementation of these methods. If your component extends a class which implements these methods, for example `fl.controls.LabelButton`, then you might want to call the `super` implementation; this is a case by case decision you'll need to make by examining the code and stepping through each line of code with the debugger.

I overrode `keyDownHandler()` to handle all keyboard events. The general form to take for a `KeyboardEvent` handler is to switch on the event's `keyCode` for special keys like escape, shift, F1-F12 and the arrow keys and match their values against the constants defined in `flash.ui.Keyboard`. To handle alphanumeric and punctuation input, you can convert the `keyCode` to a `String` with the `String.fromCharCode()` method. For

alphabetic keys this approach always returns the upper case value, and for other keys it always returns the value you would get without holding the shift key down. A simple `keyDownHandler()` might look like this:

```
override protected function keyDownHandler(e:KeyboardEvent):void {
    switch (e.keyCode) {
        case Keyboard.UP:
            trace("up");
            break;
        case Keyboard.DOWN:
            trace("down");
            break;
        case Keyboard.LEFT:
            trace("left");
            break;
        case Keyboard.RIGHT:
            trace("right");
            break;
    }
    switch (String.fromCharCode(e.keyCode)) {
        case '1':
            trace("We're number one!");
            break;
        case 'A':
            trace("A is for apple");
            break;
        case 'B':
            trace("B is for banana");
            break;
        case 'C':
            trace("C is for coconut");
            break;
    }
}
```

## Adding keyboard support to MenuBar

### Enabling selectable

Up until this point, our MenuBar component has relied completely on mouse events to open and close menus. The component currently has no way of tracking or displaying a menu item selected with the keyboard. To remedy this, I enabled selection by setting the `selectable` property to `true` on the `MenuBarTileList` instance and all of the `MenuItem` instances, undoing one of the very first changes I had made when creating the prototype in Part 1 of this article series.

Rather than making the selection persistent, I only wanted it to last as long as the keyboard was active in the menu bar or in the specific drop-down menu. Because of this, I left the code in place that set `selectable=false` and only set the value to `true` when the controls become active, then immediately setting the value back to `false` when they are closed. This part was easy to implement, since I already had methods to

manage hiding and showing the drop-down menus, as well as opening and closing the menu bar completely. You can see the changes I made to the code below.

```
private function openMenuBar(menuToOpen:List):void {
    // enable selectable for keyboard support
    myMenuBar.selectable = true;

    // open the List drop down menu
    hideAllMenusExcept(menuToOpen);

    ...
}

private function closeMenuBar():void {
    // close all menus
    hideAllMenusExcept(null);
    // reset the state of keepMenuOpen, just to make sure it isn't left funky
    keepMenuOpen = false;
    // disable selection when menu bar closed
    myMenuBar.selectedIndex = -1;
    myMenuBar.selectable = false;
    ...
}

private function hideAllMenusExcept(except:List):void {
    for (var i:int = 0; i < myMenus.length; i++) {
        var theMenu:List = myMenus[i] as List;
        if (theMenu == except) {
            theMenu.visible = true;
            theMenu.selectable = true;
        } else {
            theMenu.visible = false;
            theMenu.selectedIndex = -1;
            theMenu.selectable = false;
        }
    }
}
```

Since MenuBar does not have any selectable skins, I also changed the custom cell renderers for the menu bar and the drop-down menus to use the over skins instead. I copied the code from the implementation of `BaseButton`, which was being used by the cell renderer classes currently, and changed it slightly. To illustrate the changes, I've included the code for both the `MenuCellRenderer` and the `BaseButton` below:

```
/*
 * MenuCellRenderer version
 */
override protected function drawBackground():void {
    var styleName:String = (enabled) ? mouseState : "disabled";
    if (selected) {
        styleName = "over";
    }
    styleName += "Skin";
    var bg:DisplayObject = background;
```

```

        background = getDisplayObjectInstance(getStyleValue(styleName));
        addChildAt(background, 0);
        if (bg != null && bg != background) { removeChild(bg); }
    }

    /*
     * BaseButton version
     */
    protected function drawBackground():void {
        var styleName:String = (enabled) ? mouseState : "disabled";
        if (selected) { styleName =
"selected"+styleName.substr(0,1).toUpperCase()+styleName.substr(1); }
        styleName += "Skin";
        var bg:DisplayObject = background;
        background = getDisplayObjectInstance(getStyleValue(styleName));
        addChildAt(background, 0);
        if (bg != null && bg != background) { removeChild(bg); }
    }
}

```

I changed `MenuBarCellRenderer` to extend `MenuCellRenderer`, because it needed the exact same code to fix `drawBackground()`. I could have copied and pasted the code into both classes, but that would have created a maintenance problem, and to follow best practices, this is really an ideal situation to leverage inheritance.

```

public class MenuBarCellRenderer extends MenuCellRenderer {

```

You may be getting tired of hearing me say this, but I couldn't have achieved this part of the development process without reading and debugging the ActionScript source for the User Interface components. Being able to copy that code and make minor changes to it was a huge time saver as well.

## keyDownHandler()

To understand the `keyDownHandler()` method and its auxiliary method, `dispatchItemSelectedEvent()`, you can read the commented code below:

```

/*
 * keyboard handling
 */
override protected function keyDownHandler(e:KeyboardEvent):void {
    // keyboard support does nothing if there are not any drop-down menus
    if (myMenus.length < 1) return;

    var theMenu>List;

    // if the menu bar is not open and up or down key was hit, then open it
    if (myMenuBar.selectedIndex < 0) {
        // enable selectable
        myMenuBar.selectable = true;
        // first switch on key to determine which drop-down menu should be
opened
        switch (e.keyCode) {
            case Keyboard.UP:

```

```

    case Keyboard.DOWN:
    case Keyboard.RIGHT:
        myMenuBar.selectedIndex = 0;
        break;
    case Keyboard.LEFT:
        myMenuBar.selectedIndex = (myMenuBar.length - 1);
        break;
    }

    // open the drop-down menu
    openMenuBar(myMenus[myMenuBar.selectedIndex]);
    theMenu = myMenus[myMenuBar.selectedIndex] as List;

    // now switch to see whether the first or last item in the drop-down menu should be selected //
    switch (e.keyCode) {
    case Keyboard.RIGHT:
    case Keyboard.LEFT:
    case Keyboard.DOWN:
        theMenu.selectedIndex = 0;
        break;
    case Keyboard.UP:
        theMenu.selectedIndex = (theMenu.length - 1);
        break;
    }

    // done!
    return;
}

// this code path is hit if the menu bar was already open
switch (e.keyCode) {
case Keyboard.UP:
    // the up key moves the drop-down menu selection up,
    // or down to the bottom if the selection is at the top
    theMenu = myMenus[myMenuBar.selectedIndex] as List;
    if (theMenu.selectedIndex <= 0) {
        theMenu.selectedIndex = (theMenu.length - 1);
    } else {
        theMenu.selectedIndex--;
    }
    break;
case Keyboard.DOWN:
    // the down key moves the drop-down menu selection down,
    // or up to the top if the selection is at the bottom
    theMenu = myMenus[myMenuBar.selectedIndex] as List;
    if (theMenu.selectedIndex < 0 || (theMenu.selectedIndex + 1) >=
theMenu.length) {
        theMenu.selectedIndex = 0;
    } else {
        theMenu.selectedIndex++;
    }
    break;
case Keyboard.LEFT:
    // the left key closes the currently opened drop-down menu
    // and opens the one immediately to its left, or if the
    // leftmost menu was open then it opens the rightmost menu

```

```

        if (myMenuBar.selectedIndex <= 0) {
            myMenuBar.selectedIndex = (myMenuBar.length - 1);
        } else {
            myMenuBar.selectedIndex--;
        }
        theMenu = myMenus[myMenuBar.selectedIndex] as List;
        hideAllMenusExcept(theMenu);
        theMenu.selectedIndex = 0;
        break;
    case Keyboard.RIGHT:
        // the right key closes the currently opened drop-down menu
        // and opens the one immediately to its right, or if the
        // rightmost menu was open then it opens the leftmost menu
        if (myMenuBar.selectedIndex < 0 || (myMenuBar.selectedIndex + 1) >=
myMenuBar.length) {
            myMenuBar.selectedIndex = 0;
        } else {
            myMenuBar.selectedIndex++;
        }
        theMenu = myMenus[myMenuBar.selectedIndex] as List;
        hideAllMenusExcept(theMenu);
        theMenu.selectedIndex = 0;
        break;
    case Keyboard.SPACE:
    case Keyboard.ENTER:
        // space or enter will dispatch a menu event if a
// drop-down menu item is selected
        dispatchItemSelectedEvent();
        closeMenuBar();
    case Keyboard.ESCAPE:
        // escape will close the menu without selecting any items
        closeMenuBar();
        break;
    }
}

private function dispatchItemSelectedEvent():void {
    // get the menu bar index and label
    var menuIndex:int = myMenuBar.selectedIndex;
    // we will not dispatch an event if no menu bar item is selected
    if (menuIndex < 0) return;
    var menuLabel:String = myMenuBar.dataProvider.getItemAt(menuIndex).label;

    // get the drop-down menu item index and label
    var theMenu:List = myMenus[myMenuBar.selectedIndex] as List;
    var itemIndex:int = theMenu.selectedIndex;
    // we will not dispatch an event if no drop-down item is selected
    if (itemIndex < 0) return;
    var itemLabel:String = theMenu.dataProvider.getItemAt(itemIndex).label;

    // dispatch the event
    dispatchEvent(new MenuEvent(MenuEvent.ITEM_SELECTED, false, false,
menuIndex, menuLabel, itemIndex, itemLabel));
}

```

## Coordinating mouse input and keyboard input

Once I had written the `keyDownHandler()` method and made other code changes to get selection working in the `MenuBar` component's subcomponents, keyboard support was working great, but it was not interacting very well with mouse support. After selecting `Control > Test Movie` and doing some tests, I noticed that if I started changing the menu selection with the keyboard and then switched to using the mouse, I saw multiple menu items in the over state. If I started navigating through the menu with the mouse and then tried to take over the control with the keyboard, the next item in the menu was not selected properly. This issue occurred because the keyboard support was driven by selection, but the mouse support did not currently interact with selection at all.

To resolve this issue, I added some code to `menuBarMouseHandler()` to make sure that mouse interactions were setting the menu bar selection. The updated version of the code is shown below, with some of the comments removed:

```
private function menuBarMouseHandler(e:MouseEvent):void {
    var cellRenderer:ICellRenderer = e.target as ICellRenderer;
    if (cellRenderer == null) return;

    switch (e.type) {
    case MouseEvent.MOUSE_DOWN:
        var theMenu:List = getChildByName(cellRenderer.data.label) as List;
        openMenuBar(theMenu);
        // see MOUSE_UP handling below for discussion of keepMenuOpen
        keepMenuOpen = true;
        // set selectedIndex to improve interaction between
        // mouse support and keyboard support
        myMenuBar.selectedIndex = cellRenderer.listData.index;
        break;
    case MouseEvent.MOUSE_OVER:
        theMenu = getChildByName(cellRenderer.data.label) as List;
        hideAllMenusExcept(theMenu);
        // set selectedIndex to improve interaction between
        // mouse support and keyboard support
        myMenuBar.selectedIndex = cellRenderer.listData.index;
        break;
    case MouseEvent.MOUSE_UP:
        // this event will only be hit on the first mouseUp after the
        // first mouseDown which opened the menus. We need to handle
        // this to prevent the stage mouseUp listener from closing the
        // menus. We could stop the stage listener from getting the
        // event at all by calling e.stopPropagation(), but since this
        // will eventually become component code that could be used in
        // an arbitrary application, it seems dangerous to stop event
        // propagation since we do not know what sort of event handling
        // the user might be coding on top of ours. So instead we use
        // the keepMenuOpen:Boolean.
        myMenuBar.removeEventListener(MouseEvent.MOUSE_UP,
menuBarMouseHandler);
        keepMenuOpen = true;
        break;
    }
}
```

```
}
```

I had to start listening for `mouseover` events on the drop-down menu `MenuItem` instances, adding the listeners in `openMenuBar()` and removing them in `closeMenuBar()` like I was already doing for other event listeners on the `MenuItem` instances. I added code to `menuItemMouseHandler()` to handle this event. I was also able to simplify the event dispatching code, by using the same private method to dispatch the `itemSelected` event for both keyboard and mouse interaction:

```
private function menuItemMouseHandler(e:MouseEvent):void {
    var cellRenderer:ICellRenderer = e.target as ICellRenderer;
    if (cellRenderer == null) return;

    switch (e.type) {
    case MouseEvent.CLICK:
        // dispatch event and close menu bar
        dispatchItemSelectedEvent();
        closeMenuBar();
        break;
    case MouseEvent.MOUSE_DOWN:
        // keep stage listener from closing the menus
        // more on use of keepMenuOpen in comment
        // for mouseUp event in menuBarMouseHandler()
        keepMenuOpen = true;
        break;
    case MouseEvent.MOUSE_OVER:
        // set selectedIndex to improve interaction between
        // mouse support and keyboard support
        var theMenu:List = cellRenderer.listData.owner as List;
        theMenu.selectedIndex = cellRenderer.listData.index;
        break;
    }
}
```

After these changes to the code were implemented, everything about the behavior of the `MenuBar` component was almost working, but I found that I was still having cosmetic problems when I started control with the mouse and then took over control with the keyboard. After making some tests, I identified the problem. The over skin would remain on a menu item as long as the mouse was hovering over it, even if I moved the selection to another menu item. I fixed this issue with another change to `MenuItemRenderer`, and it was only necessary to apply the fix in one place because I altered `MenuBarCellRenderer` to subclass this class.

```
override protected function drawBackground():void {
    var styleName:String = (enabled) ? mouseState : "disabled";
    if (selected) {
        styleName = "over";
    } else if (styleName == "over") {
        styleName = "up";
    }
    styleName += "Skin";
    var bg:DisplayObject = background;
```

```
background = getDisplayObjectInstance(getStyleValue(styleName));  
addChildAt(background, 0);  
if (bg != null && bg != background) { removeChild(bg); }  
}
```

## Where to go from here

In the last part of this article series, we'll examine how to work with compiled clips. We'll put the finishing touches on our MenuBar component by ensuring that when developers use your components their FLA files will publish quickly. We'll provide some tips and resources to help you optimize your components before distributing them and also touch on some compatibility issues that you should consider when developing components.

To learn more about how to handle events and control keyboard input with ActionScript 3.0, see the following Developer Center article:

[Introduction to event handling in ActionScript 3.0](#)

## About the author

Jeff Kameron is a computer scientist at Adobe Systems who has worked on the Flash authoring team since 2002.