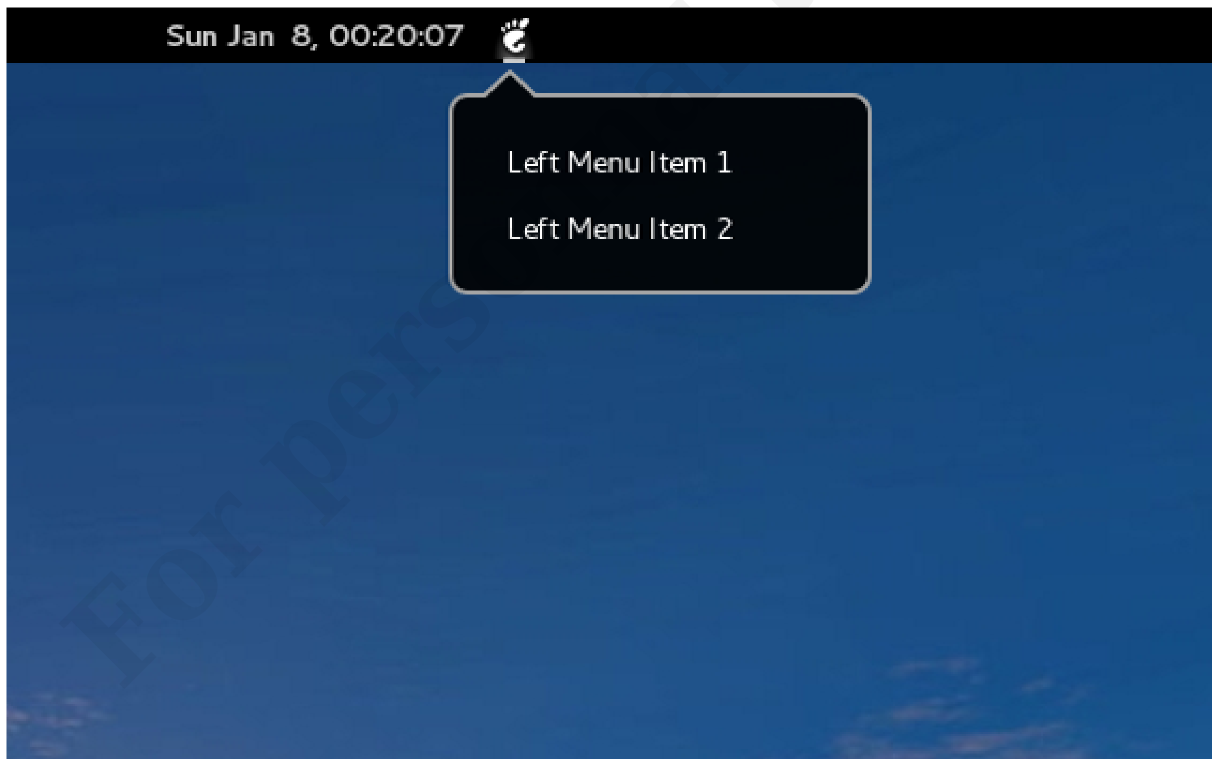
Dual Menus for a GNOME Shell 3.2 Button

Finnbarr P. Murphy
(fpm@fpmurphy.com)

One of the current limitations with the GNOME 3.2 Shell top panel buttons is that there is no support for displaying separate menus for different mouse button clicks. Yes, yes, I can hear the GNOME Shell designers say with complete authority and conviction that this use case is by design. Their answer to any criticism, constructive or otherwise, about their design is generally that they know better than anybody else and their (never-published) usability studies support their design. Just look at the *Suspend* versa *Power Off* menu option debate that erupted when the GNOME Shell was originally released!

Anyway, this particular limitation has annoyed me for a while and I recently decided to look at the issue and try and come up with a simple workable solution to the limitation.

Suppose that I wanted the following menu to be displayed when I use my left mouse button to activate the GNOME Foot button on the top panel:



and I want the following menu options to be displayed when I use my right mouse button to activate the same top panel button:

 Copyright 2004-2019 Finnbarr P. Murphy. All rights reserved.

There is nothing in the underlying Mutter code that limits support for such functionality. In fact Mutter has explicit support for returning an integer that represents a pressed or released mouse button. According to the Clutter (upon which Mutter is based) documentation, in the case of a standard scroll mouse, the following numbers are reliable:

- 1 = Left mouse button
- 2 = Scroll wheel
- 3 = Right mouse button

For mice with more buttons, you may have to experiment to see which particular button returns which particular value.

In your code, how do you determine which particular mouse button was pressed? Simply connect an event handler (AKA callback or signal handlers) to the *button-press-event* and/or the *button-release-event* signals of a suitable actor. In the event handler, use *get_button()* to return the integer representing the specific mouse button that was pressed and/or released. You can use a single function for both press and release signals.

```
...

_init: function(menuAlignment) {
     PanelMenu.ButtonBox.prototype._init.call(this,
                                          { reactive: true,
                                            can_focus: true,
                                            track_hover: true
                                          });
     ...
     this.actor.connect('button-press-event', Lang.bind(this, this._onButtonPress));
     ...
 },

 _onButtonPress: function(actor, event) {
     let button = event.get_button();
     if (button == 1) {
         // do something
```

For personnal use only

 Copyright 2004-2019 Finnbarr P. Murphy. All rights reserved.

```
        } else if (button == 3) {
            // do something
        }
    },

    ...
```

A *button-press-event* is emitted when a mouse button is pressed, but not necessarily released, on a reactive actor. A *button-release-event* is emitted when a mouse button is released on a reactive actor (even if the mouse was pressed down somewhere else beforehand). Usually you must explicitly enable the *reactive* property of an actor. By the way, a *reactive* actor (a Clutter concept) is an actor that can emit pointer events.

An alternative approach is to use *Clutter.ClickAction*.

```
   ...

   let clickAction = new Clutter.ClickAction();
   clickAction.connect('clicked', Lang.bind(this, function(button)  {
       this._onButtonPress(button);
   }));
   this.actor.add_action(clickAction);

   ...

    _onButtonPress: function(button) {
        if (button == 1) {
            // do something
        } else if (button == 3) {
            // do something
        }
    },

    ...
```

For examples of how and where *Clutter.ClickAction* is used in the GNOME Shell 3.2 code, look in */usr/share/gnome-shell/js/ui/shellEntry.js* and /usr/share/gnome-shell/js/ui/workspace.js where it is used to implement *clicked* and *long-press* event handling.

Back to the problem at hand. It turns out that the major impediment to implementing dual action menus in GNOME Shell 3.2 lies in the code that implements the Button object. See */usr/share/gnome-shell/js/ui/panelMenu.js*. Here is the relevant code:

```
function Button(menuAlignment) {
    this._init(menuAlignment);
}

Button.prototype = {
    __proto__: ButtonBox.prototype,

    _init: function(menuAlignment) {
        ButtonBox.prototype._init.call(this, { reactive: true,
                                               can_focus: true,
                                               track_hover: true });

        this.actor.connect('button-press-event', Lang.bind(this, this._onButtonPress));
        this.actor.connect('key-press-event', Lang.bind(this, this._onSourceKeyPress));
        this.menu = new PopupMenu.PopupMenu(this.actor, menuAlignment, St.Side.TOP);
        this.menu.actor.add_style_class_name('panel-menu');
```

For personnal use only

 Copyright 2004-2019 Finnbarr P. Murphy. All rights reserved.

```
        this.menu.connect('open-state-changed', Lang.bind(this, this._onOpenStateChanged))
;
        this.menu.actor.connect('key-press-event', Lang.bind(this, this._onMenuKeyPress));
        Main.uiGroup.add_actor(this.menu.actor);
        this.menu.actor.hide();
    },

function Button(menuAlignment) {
    this._init(menuAlignment);
}

Button.prototype = {
    __proto__: ButtonBox.prototype,

    _init: function(menuAlignment) {
        ButtonBox.prototype._init.call(this, { reactive: true,
                                               can_focus: true,
                                               track_hover: true });

        this.actor.connect('button-press-event', Lang.bind(this, this._onButtonPress));
        this.actor.connect('key-press-event', Lang.bind(this, this._onSourceKeyPress));
        this.menu = new PopupMenu.PopupMenu(this.actor, menuAlignment, St.Side.TOP);
        this.menu.actor.add_style_class_name('panel-menu');
        this.menu.connect('open-state-changed', Lang.bind(this, this._onOpenStateChanged))
;
        this.menu.actor.connect('key-press-event', Lang.bind(this, this._onMenuKeyPress));
        Main.uiGroup.add_actor(this.menu.actor);
        this.menu.actor.hide();
    },

   _onButtonPress: function(actor, event) {
        if (!this.menu.isOpen) {
            // Setting the max-height won't do any good if the minimum height of the
            // menu is higher then the screen; it's useful if part of the menu is
            // scrollable so the minimum height is smaller than the natural height
            let monitor = Main.layoutManager.primaryMonitor;
            this.menu.actor.style = ('max-height: ' +
                                     Math.round(monitor.height - Main.panel.actor.height)
+
                                     'px;');
        }
        this.menu.toggle();
    },

    _onSourceKeyPress: function(actor, event) {
        let symbol = event.get_key_symbol();
        if (symbol == Clutter.KEY_space || symbol == Clutter.KEY_Return) {
            this.menu.toggle();
            return true;
        } else if (symbol == Clutter.KEY_Escape && this.menu.isOpen) {
            this.menu.close();
            return true;
        } else if (symbol == Clutter.KEY_Down) {
            if (!this.menu.isOpen)
                this.menu.toggle();
            this.menu.actor.navigate_focus(this.actor, Gtk.DirectionType.DOWN, false);
            return true;
        } else
            return false;
    },

   _onMenuKeyPress: function(actor, event) {
        let symbol = event.get_key_symbol();
        if (symbol == Clutter.KEY_Left || symbol == Clutter.KEY_Right) {
            let focusManager = St.FocusManager.get_for_stage(global.stage);
            let group = focusManager.get_group(this.actor);
            if (group) {
                let direction = (symbol == Clutter.KEY_Left) ? Gtk.DirectionType.LEFT : Gt
```

For personal use only

 Copyright 2004-2019 Finnbarr P. Murphy. All rights reserved.

```
k.DirectionType.RIGHT;
                group.navigate_focus(this.actor, direction, false);
                return true;
            }
        }
        return false;
    },

    _onOpenStateChanged: function(menu, open) {
        if (open)
            this.actor.add_style_pseudo_class('active');
        else
            this.actor.remove_style_pseudo_class('active');
    },

    destroy: function() {
        this.actor._delegate = null;

        this.menu.destroy();
        this.actor.destroy();

        this.emit('destroy');
    }
};
Signals.addSignalMethods(Button.prototype);
```

If you examine the above code, you will see that no attempt is made to differentiate between the different mouse buttons. As far as the above code is concerned, one mouse button is the same as another mouse button. Perhaps the designers of the GNOME Shell grew up using a single button Apple mouse or, as many commentators have suggested, the GNOME Shell was really designed for tablets and palmhelds where no mouse is generally used or available.

The solution to implementing dual menu functionality using mouse buttons was to modify the *buttom-press-event* signal handler in the above Button code to determine which mouse button was pressed and act accordingly.

Here is the source code for a simple GNOME Shell extension that I used to prototype a working implementation of dual menu functionality:

```
const St = imports.gi.St;
const Main = imports.ui.main;
const PanelMenu = imports.ui.panelMenu;
const PopupMenu = imports.ui.popupMenu;
const Lang = imports.lang;
const Clutter = imports.gi.Clutter;
const Shell = imports.gi.Shell;
const Signals = imports.signals;
function DualActionButton(menuAlignment) {
    this._init(menuAlignment);
}
DualActionButton.prototype = {
    __proto__: PanelMenu.ButtonBox.prototype,
    _init: function(menuAlignment) {
        PanelMenu.ButtonBox.prototype._init.call(this, { reactive: true,
                                               can_focus: true,
                                               track_hover: true });
        this.actor.connect('button-press-event', Lang.bind(this, this._onButtonPress));
        this.actor.connect('key-press-event', Lang.bind(this, this._onSourceKeyPress));
        this.menuL = new PopupMenu.PopupMenu(this.actor, menuAlignment, St.Side.TOP);
        this.menuL.actor.add_style_class_name('panel-menu');
        this.menuL.connect('open-state-changed', Lang.bind(this, this._onOpenStateChanged)
);
        this.menuL.actor.connect('key-press-event', Lang.bind(this, this._onMenuKeyPress))
```

For personal use only

 Copyright 2004-2019 Finnbarr P. Murphy. All rights reserved.

```
;
        Main.uiGroup.add_actor(this.menuL.actor);
        this.menuL.actor.hide();
        this.menuR = new PopupMenu.PopupMenu(this.actor, menuAlignment, St.Side.TOP);
        this.menuR.actor.add_style_class_name('panel-menu');
        this.menuR.connect('open-state-changed', Lang.bind(this, this._onOpenStateChanged)
);
        this.menuR.actor.connect('key-press-event', Lang.bind(this, this._onMenuKeyPress))
;
        Main.uiGroup.add_actor(this.menuR.actor);
        this.menuR.actor.hide();
    },
    _onButtonPress: function(actor, event) {
        let button = event.get_button();
        if (button == 1) {
            if (this.menuL.isOpen) {
                this.menuL.close();
            } else {
                if (this.menuR.isOpen)
                    this.menuR.close();
                this.menuL.open();
            }
        } else if (button == 3) {
            if (this.menuR.isOpen) {
                this.menuR.close();
            } else {
                if (this.menuL.isOpen)
                    this.menuL.close();
                this.menuR.open();
            }
        }
    },
    _onButtonPress: function(actor, event) {
        let button = event.get_button();
        if (button == 1) {
            if (this.menuL.isOpen) {
                this.menuL.close();
            } else {
                if (this.menuR.isOpen)
                    this.menuR.close();
                this.menuL.open();
            }
        } else if (button == 3) {
            if (this.menuR.isOpen) {
                this.menuR.close();
            } else {
                if (this.menuL.isOpen)
                    this.menuL.close();
                this.menuR.open();
            }
        }
    },
    _onSourceKeyPress: function(actor, event) {
        let symbol = event.get_key_symbol();
        if (symbol == Clutter.KEY_space || symbol == Clutter.KEY_Return) {
            if (this.menuL.isOpen) {
                this.menuL.close();
            } else if (this.menuR.isOpen) {
                this.menuR.close();
            }
            return true;
        } else if (symbol == Clutter.KEY_Escape) {
            if (this.menuL.isOpen)
                this.menuL.close();
            if (this.menuR.isOpen)
                this.menuR.close();
            return true;
        } else
```

For personal use only

 Copyright 2004-2019 Finnbarr P. Murphy. All rights reserved.

```
            return false;
    },
    _onMenuKeyPress: function(actor, event) {
        let symbol = event.get_key_symbol();
        if (symbol == Clutter.KEY_Left || symbol == Clutter.KEY_Right) {
            let focusManager = St.FocusManager.get_for_stage(global.stage);
            let group = focusManager.get_group(this.actor);
            if (group) {
                let direction = (symbol == Clutter.KEY_Left) ? Gtk.DirectionType.LEFT : Gt
k.DirectionType.RIGHT;
                group.navigate_focus(this.actor, direction, false);
                return true;
            }
        }
        return false;
    },
    _onOpenStateChanged: function(menu, open) {
        if (open)
            this.actor.add_style_pseudo_class('active');
        else
            this.actor.remove_style_pseudo_class('active');
    },
    destroy: function() {
        this.actor._delegate = null;
        this.menuL.destroy();
        this.menuR.destroy();
        this.actor.destroy();
        this.emit('destroy');
    },
};
Signals.addSignalMethods(DualActionButton.prototype);
function DemoDualActionButton() {
   this._init();
}
DemoDualActionButton.prototype = {
    __proto__: DualActionButton.prototype,
    _init: function() {
        DualActionButton.prototype._init.call(this, 0.0);
        this._iconActor = new St.Icon({ icon_name: 'start-here',
                                        icon_type: St.IconType.SYMBOLIC,
                                        style_class: 'system-status-icon' });
        this.actor.add_actor(this._iconActor);
        this.actor.add_style_class_name('panel-status-button');
        let item = new PopupMenu.PopupMenuItem(_("Left Menu Item 1"));
        this.menuL.addMenuItem(item);
        item = new PopupMenu.PopupMenuItem(_("Left Menu Item 2"));
        this.menuL.addMenuItem(item);
        item = new PopupMenu.PopupMenuItem(_("Right Menu Item 1"));
        this.menuR.addMenuItem(item);
        item = new PopupMenu.PopupMenuItem(_("Right Menu Item 2"));
        this.menuR.addMenuItem(item);
        item = new PopupMenu.PopupMenuItem(_("Right Menu Item 3"));
        this.menuR.addMenuItem(item);
        item = new PopupMenu.PopupMenuItem(_("Right Menu Item 4"));
        this.menuR.addMenuItem(item);
    },
    enable: function() {
        Main.panel._centerBox.add(this.actor, { y_fill: true });
        Main.panel._menus.addMenu(this.menuL);
        Main.panel._menus.addMenu(this.menuR);
    },
    disable: function() {
        Main.panel._centerBox.remove_actor(this.actor);
        Main.panel._menus.removeMenu(this.menuL);
        Main.panel._menus.removeMenu(this.menuR);
    }
};
function init() {
```

For personal use only

 Copyright 2004-2019 Finnbarr P. Murphy. All rights reserved.

```
    return new DemoDualActionButton();
}
```

The *DualActionButton* object code is simply a modified version of the *Button* code. Adding support for a center mouse button menu is trivial; I will leave that for you to do if you need it.

Note that I do not include a comma after the last name/value pair. According to the Mozilla JavaScript documentation, Douglas Crockford and others, this is the correct syntax. Unfortunately, many Shell extensions developers appear not to know the correct syntax for object literals and include the comma since *gjs*, which is based on the Mozilla SpiderMonkey JavaScript engine and the GObject introspection framework, does not complain about this particular syntax error.

For those readers who do not like to encapsulate the GNOME Shell extension *enable* and *disable* methods, and there are quite a few of you according to comments and email that I receive, here is the source code for function based initialization, enabling and disabling of the above extension.

```
let button;
function init() {
    button = new DemoDualActionButton();
}
function enable() {
    Main.panel._centerBox.add(button.actor, { y_fill: true });
    Main.panel._menus.addMenu(button.menuL);
    Main.panel._menus.addMenu(button.menuR);
}
function disable() {
    Main.panel._centerBox.remove_actor(button.actor);
    Main.panel._menus.removeMenu(button.menuL);
    Main.panel._menus.removeMenu(button.menuR);
}
```

Some people claim that the above code for handling a Shell extension is simpler and easier to understand than the object literal coding style that I generally use. My answer is that the function approach requires you to use at least one variable that has extension scope and, often times, many such variables in more complex Shell extensions. The currently non-standard Mozilla JavaScript *let* keyword limits a variable's lexical scope to the block in which the variable is defined as well as any inner blocks contained inside the *let* block itself. However, if the *var* keyword is used instead of the *let* keyword or no keyword is used, the scope of the variable defaults to global. Global variables in JavaScript should be avoided where possible because of the potential side effects and errors they can introduce. This type of scoping error can be difficult to find and so I choose to avoid the issue entirely. Read Appendix A (Awful Parts) of Douglas Crockford's seminal book *JavaScript: The Good Parts* if you do not understand why JavaScript variables with global scope are evil.

By the way, I am not sure if the concept of a dual action menu button would breach computer accessibility (a11y) guidelines for the GNOME Shell but given that the paradigm is common in Microsoft Windows, I suspect not.

I have created a simple GNOME Shell extension, *demodualmenubutton*, based on the above code which you can download from my GNOME Shell extensions website.

Enjoy!

For personal use only

 Copyright 2004-2019 Finnbarr P. Murphy. All rights reserved.

For personal use only

 Copyright 2004-2019 Finnbarr P. Murphy. All rights reserved.