The Emerald Wave Lua Framework

User Guide

Contents

1. Introduction

The Emerald Wave Lua Framework consists of a set of classes that allow the programmer to write Lua script to interact with a user on either a TI-nspire handheld calculator or within a web browser. The programmer may add Lua script to communicate with the front end of the EW Framework. The backend of the EW Framework handles communication with either the TI-nspire handheld calculator API or the Emerald Wave ndlink module for web browser applications.

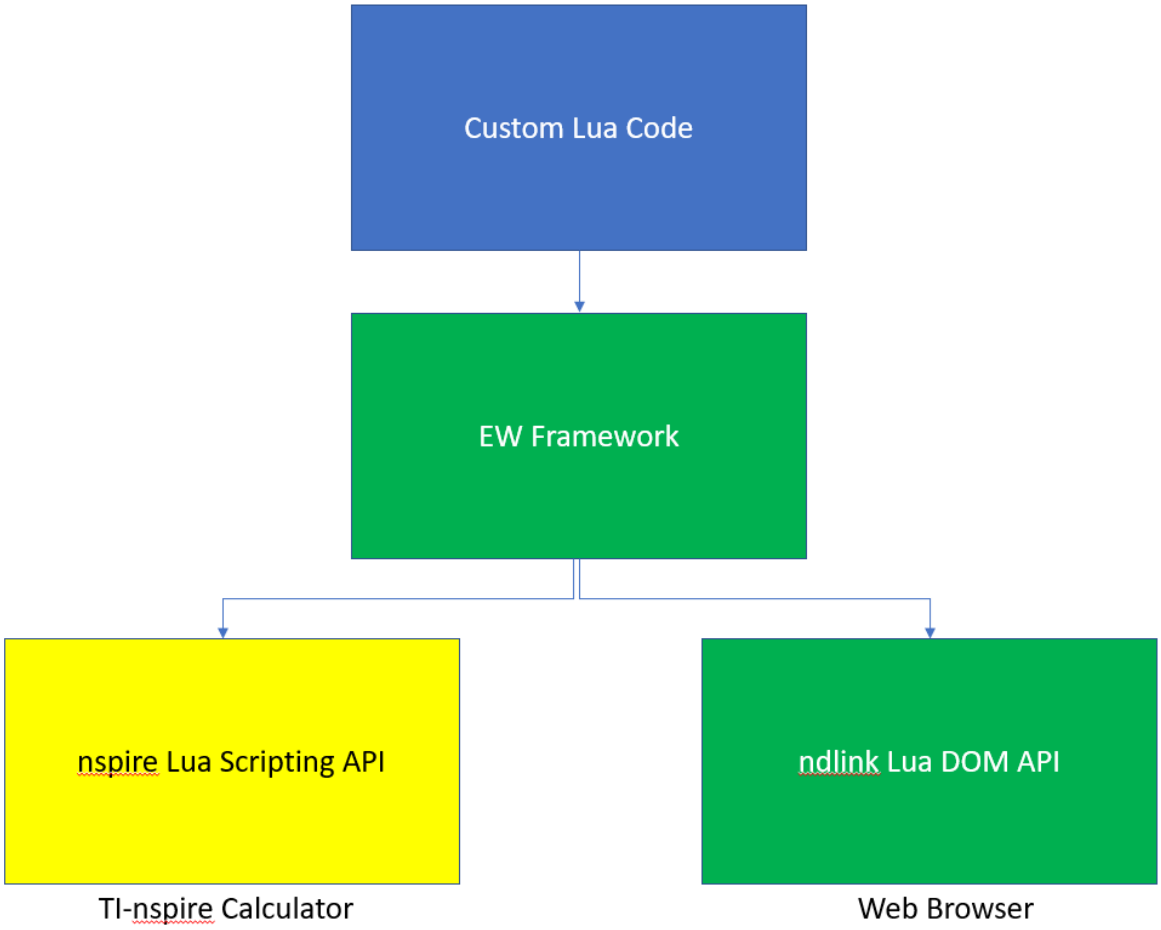

Figure 1 - EW Framework

2. Model-View-Controller

The EW Framework is based on the model-view-controller paradigm. The **app** table is the topmost table in the EW Framework and all other tables, including custom code, are accessed via subtables. Internally, the framework uses app.model for data, app.frame for user interface and events, and app.controller for logic and flow. The programmer extends the framework via one of three

programmer-supplied classes: app.model.moduleModel for data, app.frame.moduleView for user interface and events, and app.controller.moduleController for logic and flow control. In addition, the programmer supplies the view and logic pages. This is shown in Figure 2.

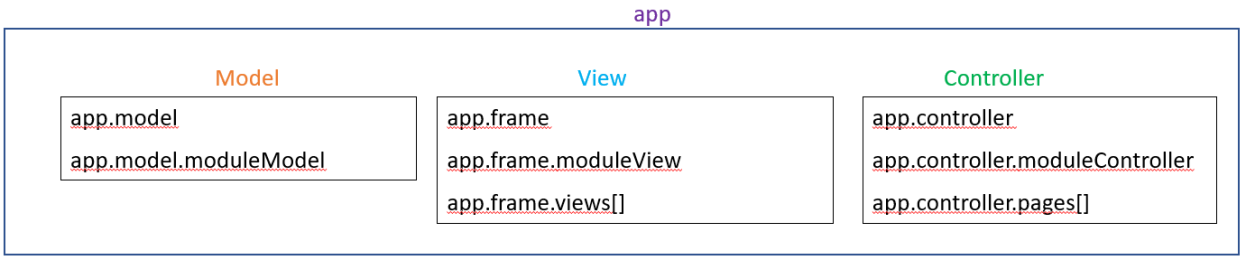

Figure 2 - Model-View-Controller

3. The App Table

The **app** table is the highest-level table in the Emerald Wave Framework. All other tables in the framework are subtables of the app table. The framework table hierarchy is shown in Figure 3. moduleModel, moduleView, and moduleController are programmer supplied tables. Custom logic pages and custom view pages are programmer supplied tables whose names are customizable in moduleModel.

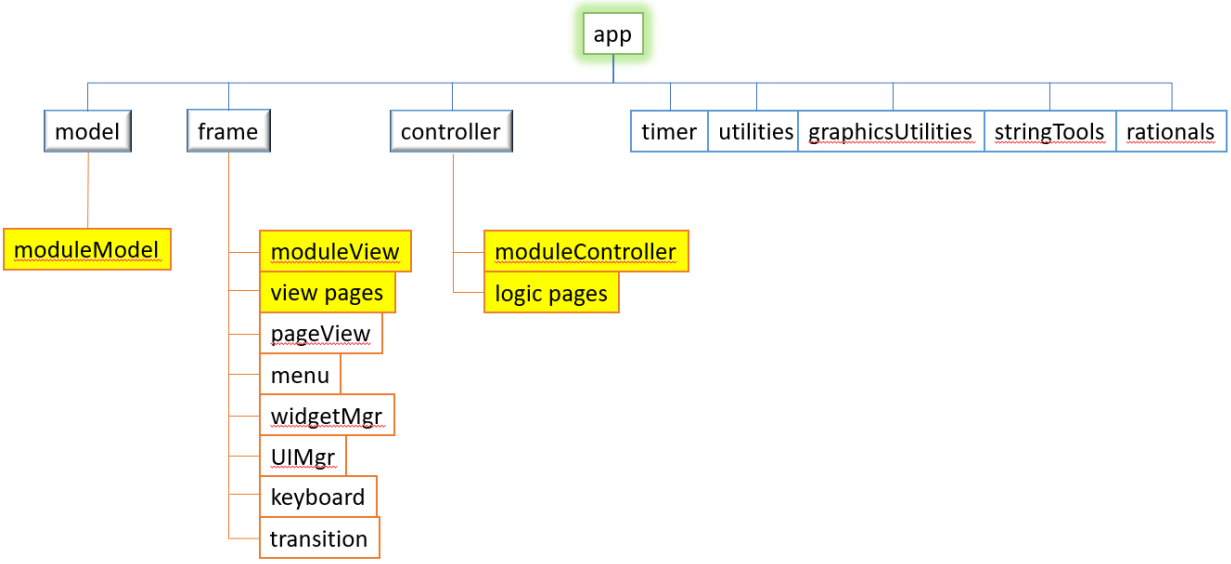

Figure 3 - Framework Tables

4. The Programmer Classes

The framework will run without any custom code from the programmer. However, to make use of the framework, the programmer creates five classes: ModuleModel, ModuleView, ModuleController, MainPage, and MainPageView, where MainPage and MainPageView class names are customizable by the programmer. The programmer may also add as many additional logic page/view page pairs and other classes as desired. Figure 4 shows the flow of the startup process.

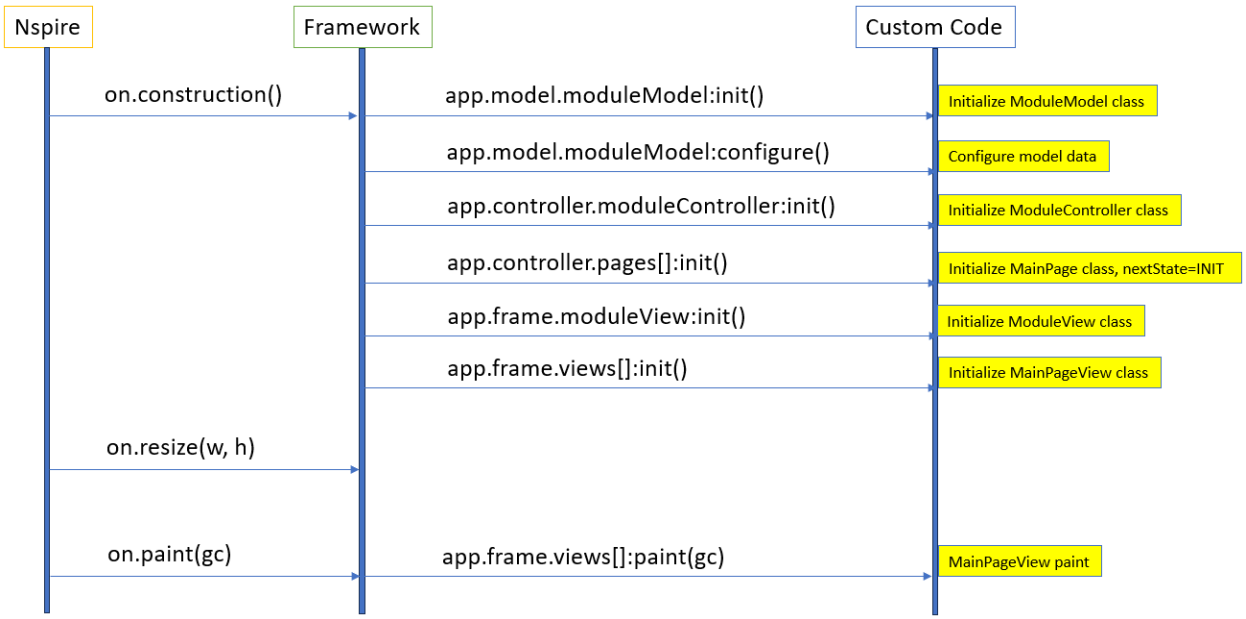

Figure 4 - Startup Flow

The example below shows the basic classes needed to utilize the framework.

ModuleModel – This class extends the framework model and allows the programmer to create custom data. Within ModuleModel, the programmer must create at least one logic page and associated view by calling app.model:setPageIDs() to create an index, and app.model:setPageList() with the list of pages used in the module. For example, to create a single page called MAIN, the ModuleModel class would contain the function configure() and look like this:

```
1. ModuleModel = class()
2.
3. function ModuleModel:init() end
4.
5. function ModuleModel:configure()
6.     app.model:setPageIDs({"MAIN_PAGE"})
7.     app.model:setPageList({[app.model.pageIDs.MAIN_PAGE] =
   {MainPage, MainPageView, "MAIN"}})
8. end
```

ModuleView – This class is created by the programmer for user interface and display. In this example, the ModuleView class does no useful work.

```
1. ModuleView = class()
2.
3. function ModuleView:init() end
```

ModuleController – This class is created by the programmer to control logic flow. In this example, handleMenu() exists to handle a menu selection, but takes no additional action.

```
1. ModuleController = class()
2.
3. function ModuleController:init() end
4. function ModuleController:handleMenu(menuID, submenuID) end
```

MainPage - The page logic class would minimally create the class and set the initial state for the dispatcher. Details about the dispatcher are explained in the next section.

```
1. MainPage = class()
2.
3. function MainPage:init(name)
4.    self.name = name
5.    self.nextState = app.model.statesList.INIT
6. end
```

MainPageView - The page view class would minimally initialize and paint.

```
1. MainPageView = class()
2.
3. function MainPageView:init(name)
4.     app.frame.pageView:initPageView(self, name)
5. end
6.
7. function MainPageView:paint(gc)
8.     gc:setColorRGB( 0, 0, 255 )
9.     gc:drawString("Hello, World!", 120, 80)
10. end
```

Figure 5 shows the screen output after the programmer has supplied the five programmer classes.

Figure 5 - Basic Programmer Classes Output

5. Dispatcher

The framework logic flow uses a state machine concept, and this is coded within the framework Controller class as a dispatcher system. The programmer sets the initial state in the startup logic page. The framework then eventually calls the dispatcher function, which enters a loop. The dispatcher calls the function specified by the programmer for the current state. After returning from the programmer code, the dispatcher checks the nextState and nextSubState properties of the current logic page. If nextState or nextSubState has changed, then the dispatcher continues to loop, and calls the function associated with the next state/substate. If the nextState and nextSubState properties are the same as the current state, the dispatcher will exit the loop, and control will be returned to the module and eventually back to the operating system to wait for a user input event or a timer event. The dispatcher is shown in Figure 6.

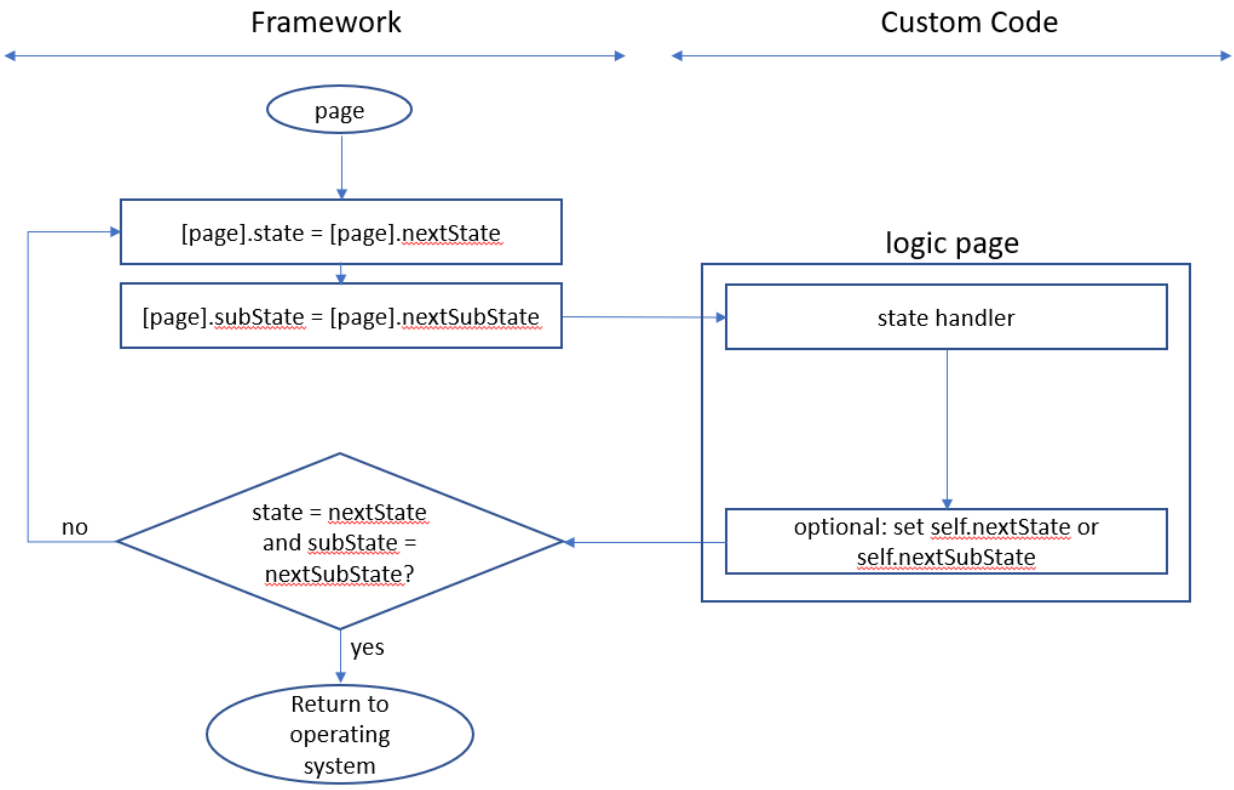

Figure 6 - Dispatcher

At startup, the framework switches to the startup page, which is defaulted as the first page in the page list. At the conclusion of the switch pages process, the framework enters the dispatcher loop with the startup page. The programmer sets the nextState property in the init() function of the startup page. The dispatcher then calls the startup page handler for the current state. The programmer may direct the dispatcher to step through additional states or allow the dispatcher to exit and return to the operating system to allow user input and timer events.

The framework contains five built-in states which are LAUNCH, INIT, END, TRANSITION, and TRANSITION_END. For these framework states, the framework will not call the logic page, but instead will call the ModuleController class for handling. The following are the names of the programmer-supplied functions that will be called for the built-in states. All states are optional.

LAUNCH – This state is intended to be used for once for the lifetime of the app. The dispatcher will call app.controller.moduleController:stateLaunch(page).

INIT – This state is intended to be used to reset a page to its initial state, such as when switching pages. The dispatcher will call app.controller.moduleController:stateInit(page).

END - This state is intended to be used at the end of a logical grouping of user actions, such as the completion of a problem, and allows the programmer an opportunity to prepare for a transition to the next set of logic operations or a different page. The dispatcher will call app.controller.moduleController:stateEnd(page).

TRANSITION – This state is intended to be used between the end of one set of logical operations and the start of a new set when the programmer wishes to display an intermediate message. The dispatcher will call app.controller.moduleController:stateTransition(page). If switching pages, the page used for painting during the transition is the current page, not the next page, and thus the decision as to what information to display is coded in the current page. The duration of the transition is controlled by a timer.

TRANSITION_END – This state is exclusively used by the framework controller transition timer to end the transition. The dispatcher will call app.controller.moduleController:stateTransitionEnd(page).

The programmer may also define custom states by calling app.model:setStatesList(). It is important to note that this call overrides the default states list, so the programmer must include all states that are used in the app. Handlers for the built-in states LAUNCH, INIT, END, TRANSITION, and TRANSITION_END, are all pre-defined in the framework. For custom states, the programmer will need to define at least one substate and must call app.model:setStateFunctions() to define a handler for a given state and substate.

Substates are only used for programmer-defined states and are not used for any of the framework built-in states. For each custom state, the logic can flow through multiple substates. The dispatcher will call the handler for the state and substate that was defined in the call to app.model.setStateFunctions(). The following are the framework built-in substates.

START – This substate is intended to be used at the start of each logical sequence of actions.

USER_INPUT – This substate is intended to be used when the user is providing input.

PROCESS_INPUT – This substate is intended to be used when the user directs the application to process the input.

WAIT_FOR_NEXT – This substate is intended to be used after the user input has been processed, but before proceeding to the next set of logic steps.

The programmer may define their own substates by calling app.model.setSubStatesList(). This call will override the framework list of default substates.

Figure 7 shows the startup process with the framework dispatcher.

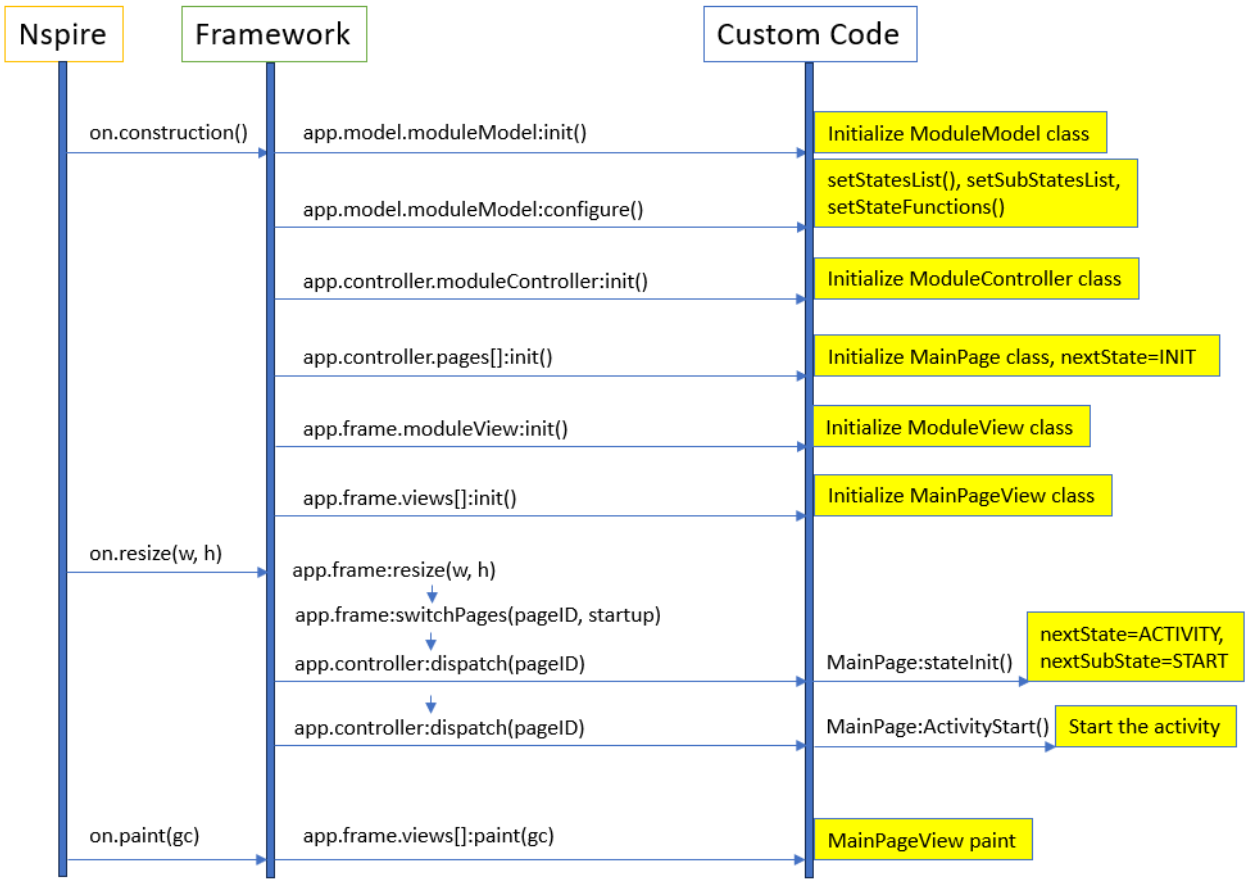

Figure 7 - Dispatcher Flow

An example of how to write the code for using states is shown below.

```
1.  function ModuleModel:configure()
2.      app.model:setStatesList({"LAUNCH", "INIT", "ACTIVITY",
    "END", "TRANSITION", "TRANSITION_END"})
3.      app.model:setSubStatesList({"START"})
4.      app.model:setStateFunctions({
          [app.model.statesList.ACTIVITY] = {
          [app.model.subStatesList.START] = function(page) return
          page:activityStart() end}, })
5.  end
6.
7.  function ModuleController:stateLaunch(page) if page.stateLaunch
    then page:stateLaunch() end end
8.  function ModuleController:stateInit(page) if page.stateInit then
    page:stateInit() end end
9.  function ModuleController:stateEnd(page) if page.stateEnd then
    page:stateEnd() end end
10.
11. function MainPage:init(name)
12.    self.name = name
```

```
13.     self.nextState = app.model.statesList.INIT
14. end
15. 
16. function MainPage:stateInit()
17.     self.nextState = app.model.statesList.ACTIVITY
18.     self.nextSubState = app.model.subStatesList.START
19. end
20. 
21. function MainPage:activityStart()
22.     --- This is the handler for the state = ACTIVITY and subState
    = START
23. end
```

6. Frame

The frame is the window for the entire application. There is only one frame, and the frame is a container for an optional header and optional footer. Figure 8 shows the bare frame, with no header, and no footer. The "Hello, World!" text can be painted directly onto the frame, or, more typically, a page view can be added to the frame for painting and holding widgets, which is described in the next section.

Figure 8 - Bare Frame

Figure 9 shows a basic frame with a header. The header can contain an optional menu button and optional title.

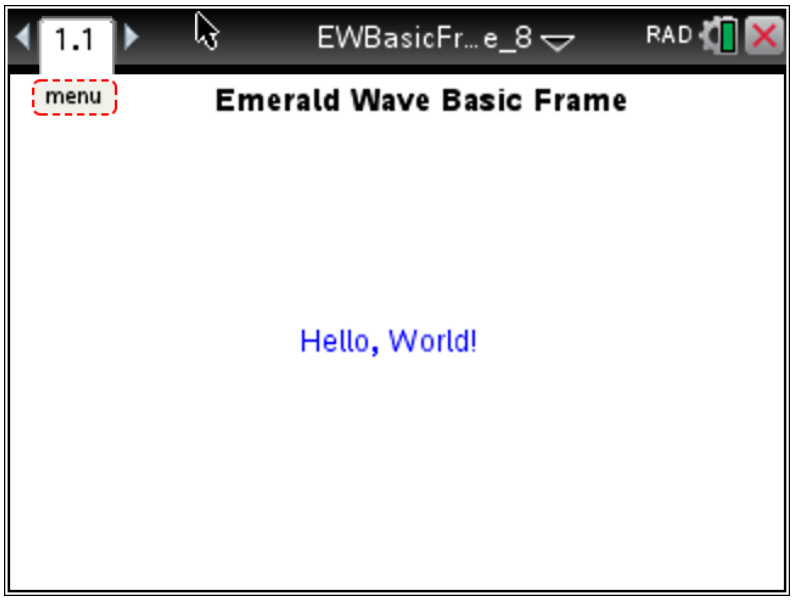

Figure 9 - Basic Frame

The following example uses the five basic programmer classes to display the Main page shown in Figure 9, and to handle a menu selection by the user.

ModuleModel

The code below defines a single page called MAIN, sets the menu and menu handler, and informs the frame that it should display the header, the footer, and a menu button, and should not display a frame border.

```
1.  ModuleModel = class()
2.
3.  function ModuleModel:init() end
4.
5.  function ModuleModel:configure()
6.      app.model:setPageIDs({"MAIN_PAGE"})
7.      app.model:setPageList({[app.model.pageIDs.MAIN_PAGE] =
    {MainPage, MainPageView, "MAIN"}})
8.      app.model:setMenuIDs({"MAIN_MENU"})
9.      app.model:setSubMenuIDs({"SUB_MENU"})
10.    app.model:setMenu({{"Main", {"Main", function()
    app.controller.moduleController:handleMenu(1, 1) end}, }, })
11.     app.model:setFrameBorder(false)
12.     app.model:setShowHeader(true)
13.     app.model:setShowFooter(true)
14.     app.model:setShowMenuButton(true)
15. end
```

ModuleView

There is nothing for ModuleView to do in this example, so this code is not technically required.

```
1. ModuleView = class()
2.
3. function ModuleView:init() end
```

ModuleController

The function to handle the menu button selections is ModuleController:handleMenu(), which receives a menuID and subMenuID once the user selects a menu and submenu item. The call to app.frame:disableMenuArrow() will ensure that the animated yellow arrow is disabled. The programmer must notify the frame of the selected menu options by calling app.frame:setMenuID() and app.frame:setSubMenuID(). The programmer can then act based on the user selection, such as by calling a function for a given page. In the example below, the menu handler will call the function setMessageID().

```
1.  ModuleController = class()
2.
3.  function ModuleController:init() end
4.
5.  function ModuleController:stateLaunch(page) if page.stateLaunch
    then page:stateLaunch() end end
6.  function ModuleController:stateInit(page) if page.stateInit then
    page:stateInit() end end
7.  function ModuleController:stateEnd(page) if page.stateEnd then
    page:stateEnd() end end
8.
9.  function ModuleController:handleMenu(menuID, subMenuID)
10.     local page =
    app.controller:getPage(app.model.pageIDs.MAIN_PAGE)
11.
12.     app.frame:disableMenuArrow()
13.     app.frame:setMenuID(menuID)
14.     app.frame:setSubMenuID(subMenuID)
15.     page:setMessageID()
16. end
```

MainPage

During the INIT state, the ModuleController will call the logic page function called stateInit(). The page logic can store the view into self.view for future reference. The viewID and pageID are one-to-one. The page logic stateInit() should then call the corresponding stateInit() function in the view to initialize user interface objects. In this example, the menu handler is called setMessageID() and will be called by ModuleController in response to the user selecting a menu/submenu item. The page logic will then call the function setMessageID(2) to tell the view to display a different message.

```
1.  MainPage = class()
2.
3.  function MainPage:init(name)
4.      self.name = name
```

```
5.      self.nextState = app.model.statesList.INIT
6.  end
7.
8.  function MainPage:stateInit()
9.      self.view = app.frame:getView(self.viewID)
10.     self.view:stateInit()
11. end
12.
13. function MainPage:setMessageID()
14.    self.view:setMessageID(2)
15. end
```

MainPageView

The view page in this example displays the string “Hello, World!” based on the variable self.viewMode. Once the user selects the menu/submenu item, the page logic will call the function setMessageID() in the view page to set the variable message = 2. This in turn will cause the view to paint a different message.

```
1.  MainPageView = class()
2.
3.  function MainPageView:init(name)
4.      app.frame.pageView:initPageView(self, name)
5.      self.messageID = 1
6.  end
7.
8.  function MainPageView:setMessageID(messageID)
9.     self.messageID = messageID
10. end
11.
12. function MainPageView:paint(gc)
13.     gc:setColorRGB( 0, 0, 255 )
14.     if self.messageID == 1 then gc:drawString("Hello, World!",
    120, 80) else gc:drawString("Thank you!", 120, 80) end
15. end
```

7. Layout Manager

While the programmer may paint directly onto the frame, the programmer will more likely want to use the framework user interface widgets such as textboxes and buttons. The framework contains a built-in layout manager that allows the programmer to position UI widgets within a scroll pane. The layout manager will automatically size and reposition UI widgets when the application frame is resized, such as in the TI-nspire Emulator Computer View or in a web browser.

The layout is composed of vertical y-position values and horizontal x-position values. The programmer creates a table containing y-position instructions for each object, and a separate table containing x-position instructions for each object. The positions can be referenced to the scroll pane, or

to another object. The programmer then provides two functions called layoutY() and layoutXOneColumn() that are part of the view class. These functions will be called during the layout process and must return a table containing the layout instructions.

The layout vertical y-table has the following properties:

object – the object being laid out.

anchorTo – Either the pane or another object to anchor to.

anchorPosition – One of: “FrameTop”, “FrameMiddle”, “FrameBottom”, “PaneTop”, “PaneMiddle”, “PaneBottom”, “Top”, “Middle”, or “Bottom”.

position – One of: “Top”, “Middle”, or “Bottom”.

offset – A value in pixels. A positive value shifts the object down. A negative value shifts the object up.

The layout horizontal x-table has the following properties:

object – the object being laid out.

anchorTo – Either the pane or another object to anchor to.

anchorPosition – One of: “PaneLeft”, “PaneMiddle”, “PaneRight”, “Left”, “Middle”, or “Right”.

position – One of: “Left”, “Middle”, or “Right”.

offset – A value in pixels. A positive value shifts the object right. A negative value shifts the object left.

The following example uses the five basic programmer classes to display a page with a textbox and a button as shown in Figure 10 using the Layout Manager.

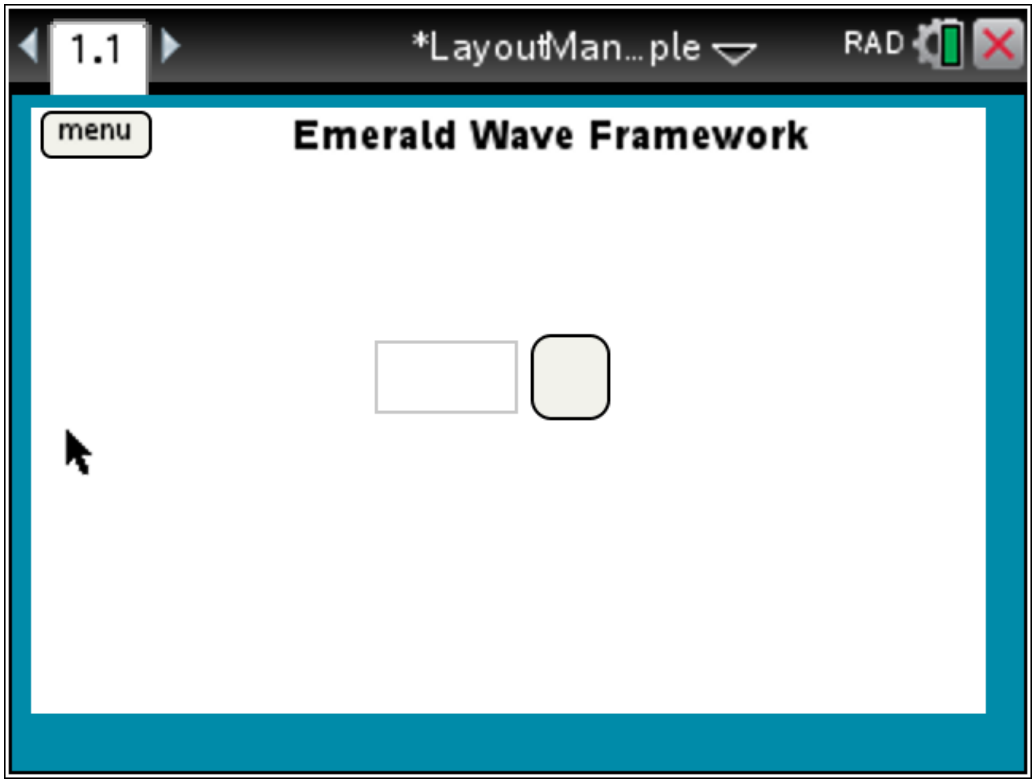

Figure 10 – Layout Manager

ModuleModel

The code below defines a single page called MAIN, sets the menu and menu handler, and informs the frame that it should display the header, the footer, and a menu button, and should not display a frame border.

```
1.  ModuleModel = class()
2.
3.  function ModuleModel:init() end
4.
5.  function ModuleModel:configure()
6.      app.model:setPageIDs({"MAIN_PAGE"})
7.      app.model:setPageList({[app.model.pageIDs.MAIN_PAGE] =
    {MainPage, MainPageView, "MAIN"}})
8.      app.model:setMenuIDs({"MAIN_MENU"})
9.      app.model:setSubMenuIDs({"SUB_MENU"})
10.    app.model:setMenu({{"Main", {"Main", function()
    app.controller.moduleController:handleMenu(1, 1) end}, }, })
11.     app.model:setFrameBorder(true)
12.     app.model:setShowHeader(true)
13.     app.model:setShowFooter(true)
14.     app.model:setShowMenuButton(true)
15. end
```

ModuleView

There is nothing for the ModuleView to do in this example, so this code is not technically required.

```
1.  ModuleView = class()
2.
3.  function ModuleView:init() end
```

ModuleController

The ModuleController handles the states and the menu.

```
1.  ModuleController = class()
2.
3.  function ModuleController:init() end
4.  function ModuleController:stateInit(page) page:stateInit() end
5.  function ModuleController:handleMenu(menuID, subMenuID) end
```

MainPage

The init() class function sets the initial state. The framework dispatcher calls stateInit() as the first state, which in turn calls self.view.stateInit() for the view.

```
1.  MainPage = class()
2.
3.  function MainPage:init(name)
4.      self.name = name
5.      self.nextState = app.model.statesList.INIT
6.  end
7.
8.  function MainPage:stateInit()
9.      self.view = app.frame:getView(self.viewID)
10.     self.view:stateInit()
11. end
```

MainPageView

The view page does all the work interacting with the Layout Manager. Instead of having a paint() function, the view configures the layout of the widgets onto the scroll pane. setupView() is called by the framework to allow the programmer the opportunity to instantiate widgets and add them to the scroll pane. The functions layoutY() and layoutXOneColumn() set the position of the textbox and button. The function stateInit() calls app.frame.pageView:updateView(self) to update the view with this layout. The Widget and the Widget Manager classes are described in a later section.

```
1.  MainPageView = class()
2.
3.  function MainPageView:init(name)
4.      app.frame.pageView:initPageView(self, name)
5.  end
6.
7.  function MainPageView:setupView()
8.     self.textbox1 = app.frame.widgetMgr:newWidget("TEXTBOX",
    self.pageID.."_textbox1", {initSizeAndPosition = {44, 22, 0,
    0}})
9.     self.scrollPane:addObject(self.textbox1)
10.
11.    self.button1 = app.frame.widgetMgr:newWidget("BUTTON",
    self.pageID.."_button1", {label = "", initSizeAndPosition =
    {24, 26, .23, 0} } )
12.    self.scrollPane:addObject(self.button1)
13. end

14. function MainPageView:layoutY()
15.    local anchorData = {}
16.    anchorData[1] = {object = self.textbox1, anchorTo =
    self.scrollPane, anchorPosition = "PaneMiddle", position =
    "Middle", offset = -20}
17.    anchorData[2] = {object = self.button1, anchorTo =
    self.scrollPane, anchorPosition = "PaneMiddle", position =
    "Middle", offset = -20}
18.
19.    return anchorData
20. end
21.
22. function MainPageView:layoutXOneColumn()
23.     local anchorData = {}
24.
25.    anchorData[1] = {object = self.textbox1, anchorTo =
    self.scrollPane, anchorPosition = "PaneMiddle", position =
    "Middle", offset = -20 }
26.    anchorData[2] = {object = self.button1, anchorTo =
    self.scrollPane, anchorPosition = "PaneMiddle", position =
    "Middle", offset = 20}
27.
28.    return anchorData
29. end
30.
31. function MainPageView:stateInit()
32.    app.frame.pageView:updateView(self)
33. end
```

8. Widgets

Widgets are user interface objects that can be placed onto the frame or onto a scroll pane. The scroll pane is also a widget. The framework contains many built-in widgets, and the programmer may code and add custom widgets. Each built-in widget is pre-registered with the Widget Manager by the framework.

Figure 11 shows a view page with the following widgets: scroll pane, a menu button, a textbox, and a check button.

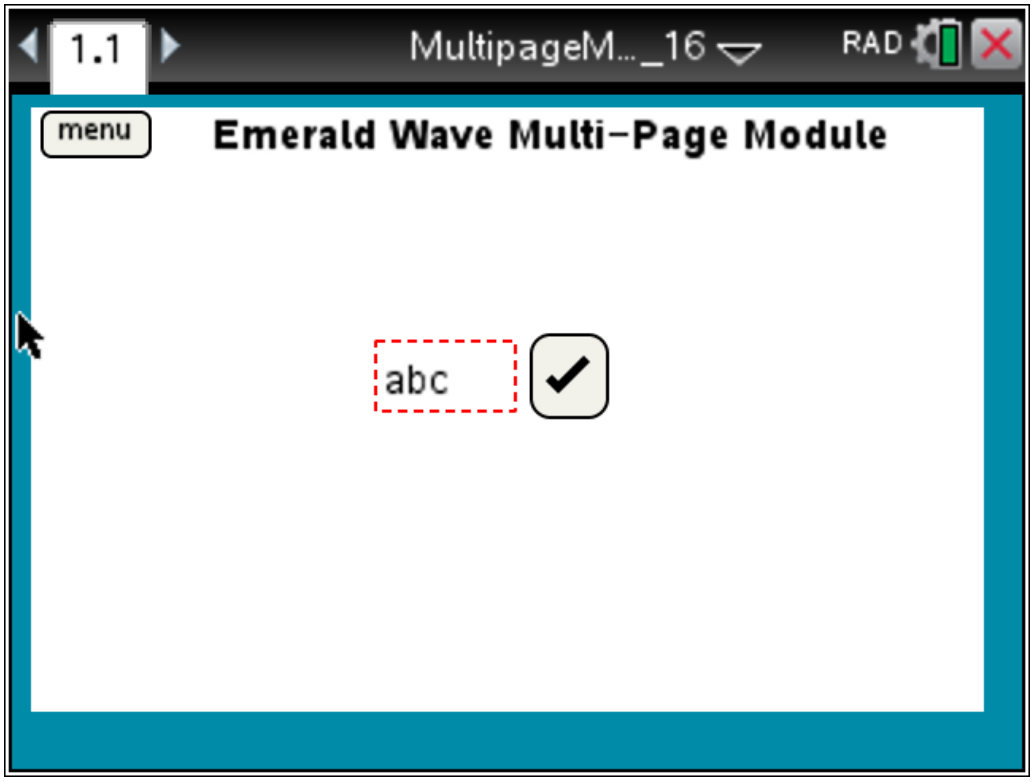

Figure 11 - Textbox and Button Widgets

The example below shows the view page programmer code that instantiates the widgets, attaches listeners, and handles events. MainPageView:setupView() is called by the framework to allow the programmer to create user interface objects. In this example, a textbox and a button widget are created by calling app.frame.widgetMgr:newWidget().

```
1.  function MainPageView:setupView()
2.      self.textbox1 = app.frame.widgetMgr:newWidget("TEXTBOX",
    self.pageID.."_textbox1", {initSizeAndPosition = {44, 22, 0, 0},
    minWidth = 44, labelPosition = "left", maxChar = 4, text = "",
    initFontAttributes = {"sansserif", "b", 13}, keyboardYPosition =
    1, allowScroll = false, isDynamicWidth = true, hasGrab = false }
    )
3.      self.textbox1:addListener(function( event, ...) return
    self:textBoxListener(event, ...) end)
```

```
4.    app.frame.UIMgr:addListener( self.pageID, function(...)
      return self.textbox1:UIMgrListener(...) end )
5.    self.scrollPane:addObject(self.textbox1)
6.
7.    self.button1 = app.frame.widgetMgr:newWidget("BUTTON",
      self.pageID.."_button1", {label = "", initSizeAndPosition = {24,
      26, .23, 0}, octagonStyle = true, drawCallback = function(...)
      app.frame.pageView:drawCheckMark(...) end } )
8.    self.button1:addListener(function( event, ...) return
      self:clickButton1(self, event, ...) end)
9.    app.frame.UIMgr:addListener( self.pageID, function(...)
      return self.button1:UIMgrListener(...) end )
10.   self.scrollPane:addObject(self.button1)
11.   end
```

Once the widget has been created and the listeners have been attached, the programmer may add the widget to the view page scroll pane via a call to self.scrollPane.addObject(widget).

Each widget type has properties that define the visual presentation and behavior of the widget. The following properties and functions are available for all widgets.

init(name) – Called during class initialization. 'name' is a unique string name for the widget being instantiated.

resize(panex, paney, panew, paneh, scaleFactor) – Called when the main window is resized. This function sets the pane, the size, and the position based on the new scale Factor. panex is the x coordinate of the upper left corner of the pane in pixels, paney is the y coordinate of the upper left corner of the pane in pixels, panew is the width of the pane in pixels, paneh is the height of the pane in pixels, scaleFactor is the current scale factor of the app.

paint(gc) – Called by the operating system to paint the widget using the provided gc.

setInitSizeAndPosition(w, h, pctx, pcty) – Sets the initial size and position of the widget on the container. w = width in pixels, h = height in pixels, pctx = percentage of the container width of the left corner of the widget, 0 to 1, pcty – percentage of the container height of the upper corner of the widget, 0 to 1.

setPane(panex, paney, panew, paneh, scaleFactor) – Sets the container pane for the widget. panex is the x coordinate of the upper left corner of the pane in pixels, paney is the y coordinate of the upper left corner of the pane in pixels, panew is the width of the pane in pixels, paneh is the height of the pane in pixels, scaleFactor is the current scale factor of the app.

setSize(w, h) – Sets the size of the widget. w is the width in pixels. h is the height in pixels.

setPosition(pctx, pcty) – Sets the position of the widget within the container as a percentage. pctx is a value from 0 to 1, referenced from the left edge of the container. pcty is a value from 0 to 1 referenced from the top edge of the container.

invalidate() – Invalidates the widget painting area to force a repaint.

contains(x, y) – Returns 1 if the x,y position is within the widget, or 0 if not.

calculateWidth(scaleFactor) – Calculates the width of the widget based on the provided scaleFactor.

calculateHeight(scaleFactor) – Calculates the height of the widget based on the provided scaleFactor.

modifyProperties(options) – 'options' is an array which contains the property name, an equals sign, and the property value. For example, {active = true, visible = false}. This is a convenience function that allows the programmer to set multiple widget properties at once.

setInitFontAttributes(fontFamily, fontStyle, fontSize) – Sets the initial fontFamily, fontStyle, and fontSize for the widget text.

setVisible(b) – Sets the visibility of the widget. b = true for visible, or b = false for not visible.

setActive(b) – Controls whether the widget is active. b = true for active, or b = false for not active.

setFocus(b) – Sets a flag that indicates if the widget should display a focus indicator when b = true. Typically, the programmer does not call this function directly because the UI Manager handles the setting and unsetting of focus.

setScrollY(y) – Sets the virtual y position of the widget on the scroll pane.

setFontColor(color) – Sets the font color for any text displayed on the widget. color is an array containing the red, blue, and green color values, each from 0 to 255.

mouseDown(x, y) – Called when the mouse down button is pressed using the x,y position of the mouse. This function will notify listeners on this widget with the event EVENT_MOUSE_DOWN if the mouse is clicked on the widget. For the return, the first parameter returns true if the mouse click is over the widget, or false if not over. For the second parameter, returns the widget object.

mouseUp(x, y) – Called when the mouse down button is released using the x,y position of the mouse. This function will notify listeners on this widget with the event EVENT_MOUSE_UP. For the return, the first parameter returns true. For the second parameter, returns the widget object.

mouseMove(x, y) – Called when the mouse is moved using the x,y position of the mouse. This function will notify listeners on this widget with the event EVENT_MOUSE_MOVE if the mouse is over the widget. If the widget is in drag mode, then this call will set the widget to its new position using the provided x,y values. For the return, the first parameter returns true if the mouse is over the widget, or false if not over. For the second parameter, returns the widget object.

mouseOutHandler() – Called when the mouse is moved off of the widget. This function will restore the widget colors to the colors used when the mouse is not over the widget.

mouseOverHandler() – Called when the mouse is moved over the widget. This function will set the widget colors to the colors used when the mouse is over the widget.

grabUp(x, y) – Called when the grab button is released, which locks the mouse cursor to the widget. x, y are the pixel positions of the pointer when the grab release occurs. Returns true if the grab is released over the widget and sets self.mouseGrabbed = true, otherwise returns false.

releaseGrab() – Called when the widget grab is released. Sets self.mouseGrabbed = false.

addListener(listener) – Adds a listener to the widget. 'listener' is a function to be called by the widget for an event. Returns a listenerID that should be stored by the caller.

removeListener(listenerID) – Removes the listener using the provided listenerID.

notifyListeners(event, …) – Called to notify all widget listeners of an event. 'event' is one of the list of events specified via app.model:registerEvents(). The '…' are any additional parameters that are needed for the event.

UIMgrListener(event, x, y) – Called by the UIMgr when an event occurs. 'event' is a registered event. x and y are the pixel positions of the mouse.

The framework built-in widgets are each derived from the Widget class. Each built-in widget has the Widget methods, plus additional methods, as described below.

Button

The Button widget allows the user to press a button with keyboard or mouse.

setDrawCallback(drawFunction) – Call this method to set a callback function for custom drawing on the button, such as a check mark.

setStyle(style) – Sets the drawing style of the button. 'style' must exist in the table self.styleIDs and can be one of RECTANGLE, ROUNDED_RECTANGLE, OCTAGON, or CIRCLE.

setLabel(label) – Sets the label text.

TextBox

The TextBox widget allows the user to enter text from a keyboard.

setLabelPosition(position) – Sets the position of the textbox label. 'position' must exist in the table self.labelPositionIDs and can be one of LEFT, TOP, TOPLEFT, or RIGHT.

clear() – Sets the text inside the textbox to an empty string.

clearAndReset() – Sets the text inside the textbox to an empty string and resets the dimensions of the textbox to their initial values.

setLabel(label1, label2) – Sets the label text. 'label1' is the typical label text. 'label2' is a second row of text, only as necessary.

setProcessCalback(fn) – Allows the programmer to pre-process any typed character by calling the provided callback function 'fn'.

setEnterKeyCalback(fn) – Allows the programmer to pre-process the Enter key by calling the provided callback function 'fn'.

setText(text) – Sets the textbox text to an initial string.

changeTextboxColor(tblTextboxColorRGB) – Sets the textbox color.

getText() – Returns the text inside the textbox.

SimpleString

The SimpleString widget displays text.

setText(text) – Sets the text to an initial string.

addUnderline(b) – 'b' is true to underline the text.

Paragraph

The Paragraph widget displays large amounts of text.

setInitViewableWindowHeight(viewableWindowHeight) – Sets the height of the viewing rectangle containing the text.

setText(text, drawMode) – Sets the paragraph text to the provided 'text' value. 'drawMode' must exist in the table self.drawModes and can be one of NO_ANIMATION, ANIMATE_CHARACTER, or ANIMATE_WORD.

addHighlight(lineNum, str, dataType, textCount, activeState, startIdx, endIdx) – Adds highlighting to the text. 'lineNum' is the text line number. 'str' is the text to display, 'dataType' must be one of self.DATA_FRACTION, self.DATA_ABSOLUTE_VALUE, self.DATA_STRING, self.DATA_FRACTION_NUMERATOR, self.DATA_FRACTION_DENOMINATOR, 'textCount' is typically equal to 1, but a higher value would be the order in the line for repeated text, 'activeState' is an element from app.model.statesList, 'startIdx' and 'endIdx' are the start and end indexes within the highlighted text to allow for highlighting just a section of the text.

setHighlightColor(highlightIdx, r, g, b) – Sets the color for a highlighted item. 'highlightIdx' is the index into the array of highlights created by addHighlight. r, g, and b are the red, green, and blue color values.

DropDownBox

The DropDownBox widget allows the user to select text from a dropdown menu.

setItems(tblItems) – Sets the list of items displayed in the dropdown box.

changeFillColor(colorRGB) – Sets the dropdown box fill color. 'colorRGB' is an array of red, blue, green values.

setDrawStyle(style) – 'style' is one of app.graphicsUtilities.drawStyles.OUTLINE_ONLY, app.graphicsUtilities.drawStyles.FILL_AND_OUTLINE, or app.graphicsUtilities.drawStyles.FILL_ONLY

clear() – Empties the dropdown box contents.

getWidthOfWidestItem() – Returns the pixel width of the widest text item in the dropdown box.

setProcessCalback(fn) – Allows the programmer to pre-process any typed character by calling the provided callback function 'fn'.

getValue() – Returns the selected item.

SimpleGrid

The SimpleGrid widget displays a table of text.

setRowsAndColumns(rows, columns) – Sets the number of rows and columns. 'rows' is the number of rows. 'columns' is the number of columns.

setBorderColor(color) – Sets the grid border color. 'color' is an array of red, green, blue values.

setCellColor(row, column, color) – Sets the color for an individual cell. 'row' is the row number. 'column' is the column number. 'color' is an array of red, green, blue values.

setCellObject(row, column, obj, pos) – Sets a widget to display for an individual cell. 'row' is the row number. 'column' is the column number. 'obj' is the widget to display. 'pos' is one of self.objectPositionID.X_LEFT, self.objectPositionID.X_MIDDLE, self.objectPositionID.X_RIGHT, self.objectPositionID.Y_TOP, self.objectPositionID.Y_MIDDLE, self.objectPositionID.Y_BOTTOM.

removeCellObject(row, column) – Removes a widget from an individual cell. 'row' is the row number. 'column' is the column number.

removeAllCellObjects() – Removes all widgets from the grid.

findCellOfObject(obj) – Returns the row and column of the provided widget. Returns nil if the object is not found.

getRowHeight() – Returns the pixel height of a row.

getColumnWidth() – Returns the pixel width of a column.

setColumnWidthPct(col, pctw) – For a given column, sets the pixel width of the column based on the percentage of the entire width of the grid. 'col' is the column number. 'pctw' is a decimal value from 0 to 1.

setRowHeightPct(row, pcth) – For a given row, sets the pixel height of the row based on the percentage of the entire height of the grid. 'row' is the row number. 'pcth' is a decimal value from 0 to 1.

getCellSizePct(a, b) – Returns the percentage of the height and the percentage of the width of the entire grid. 'a' is the cell row number, 'b' is the cell column number.

setCellText(row, column, text, pos, font) – Sets the text to display for an individual cell. 'row' is the row number. 'column' is the column number. 'text' is the text to display. 'pos' is one of self.objectPositionID.X_LEFT, self.objectPositionID.X_MIDDLE, self.objectPositionID.X_RIGHT, self.objectPositionID.Y_TOP, self.objectPositionID.Y_MIDDLE, self.objectPositionID.Y_BOTTOM. 'font' is an array of fontFamily, fontStyle, and fontSize.

setFont(fontFamily, fontStyle, initFontSize) – Sets the font for display of text in a cell. The provided values are from the nSpire font values.

setDrawGridBorder(b) – 'b' is true to draw a border for the grid.

Fraction

The Fraction widget displays a fraction with numerator, horizontal bar, and denominator. This class also performs fraction calculations.

setAbsoluteVisible(b) – 'b' is true to draw absolute value vertical bars.

setNegativesVisible(overallNegative, numeratorNegative, denominatorNegative) – 'overallNegative' is true to show the negative sign for the entire fraction. 'numeratorNegative' is true to show the negative sign for the numerator. 'denominatorNegative' is true to show the negative sign for the denominator.

setFraction(f) – 'f' is a string such as "-3/4" or "|1/2|".

splitNumeratorAndDenominatorFromString(s) – 's' is a string such as "-3/4". Returns the numerator and denominator as number values.

add(f2) – Adds the fraction 'f2' to the existing fraction, creates a new fraction, and returns the new fraction object. 'f2' is a Fraction object.

subtract(f2) – Subtracts the fraction 'f2' from the existing fraction, creates a new fraction, and returns the new fraction object. 'f2' is a Fraction object.

multiply(f2) – Multiplies the fraction 'f2' by the existing fraction, creates a new fraction, and returns the new fraction object. 'f2' is a Fraction object.

divide(f2) – Divides the existing fraction by the fraction 'f2', creates a new fraction, and returns the new fraction object. 'f2' is a Fraction object.

isGreaterThan(f2) – Returns true if the existing fraction is greater than the provided fraction. 'f2' is a Fraction object.

toDecimal() – Returns the fraction as a decimal number.

toString() – Returns the fraction as a string, such as "1/2"

GCD(a, b) – Returns the Greatest Common Divisor of a and b.

LCM(a, b) – Returns the Least Common Multiple of a and b.

MixedNumberInput

The MixedNumberInput widget accepts a whole number, fraction, or mixed number via the keyboard.

setEnterKeyCallback(cf) – Allows the programmer to pre-process the Enter key by calling the provided callback function 'cf'.

setFocusToFirstObj() – Sets focus to the first available focus element within the mixed number input boxes.

setFocusToLastObj() – Sets focus to the last available focus element within the mixed number input boxes.

setIsDynamicWidth(b) – 'b' is true for allowing the width of input boxes to expand.

setHasWholeNumberInput(b) – 'b' is true for configuring the widget to accept whole numbers.

setHasFractionInput(b) – 'b' is true for configuring the widget to accept proper fractions.

setHasUnitInput(b) – 'b' is true for configuring the widget to accept units of measure.

setFractionInputFill(b1, b2) – 'b1' is true for the numerator to fill with color. 'b2' is true for the denominator to fill with color.

setWholeNumberFill(b) – 'b' is true for the whole number input box to fill with color.

setUnitsFill(b) – 'b' is true for the units of measure input box to fill with color.

setBorderType(borderType) – 'borderType' is either "dashed" or "smooth".

setMaxChar(maxChar) – 'maxChar' is the maximum number of characters for each input textbox.

getValue(shouldCheckForVisibility) – Returns the mixed number input as two strings. The first string contains the mixed number such as "1 2/3". The second string contains the units, such as "feet". If 'shouldCheckForVisibility' is false, then the return will include the fraction part. If 'shouldCheckForVisibility' is true, then the return will include the fraction part only if there is a fraction input box.

getWholeNumberValue() – Returns the whole number.

getNumeratorValue() – Returns the numerator.

getDenominatorValue() – Returns the denominator.

getUnitValue() – Returns the units of measure.

clear() – Empties all the input text boxes.

clearWholeNumberInput() – Empties all the whole number input text box.

clearFractionInput() – Empties all the numerator and denominator input text boxes.

clearUnitInput() – Empties all the units input text box

changeWholeNumberInputColor(tblTextboxColorRGB1) – Sets the whole number input textbox color. 'tblTextboxColorRGB1' is an array of red, green, blue values.

changeFractionInputColor(tblTextboxColorRGB1) – Sets the numerator and denominator input textboxes color. 'tblTextboxColorRGB1' is an array of red, green, blue values.

changeUnitInputColor(tblTextboxColorRGB1) – Sets the units input textbox color. 'tblTextboxColorRGB1' is an array of red, green, blue values.

changeColor(tblTextboxColorRGB1) – Sets the color for all input textboxes. 'tblTextboxColorRGB1' is an array of red, green, blue values.

setDrawStyle(style) – 'style' is one of app.graphicsUtilities.drawStyles.OUTLINE_ONLY, app.graphicsUtilities.drawStyles.FILL_AND_OUTLINE, or app.graphicsUtilities.drawStyles.FILL_ONLY

setUnitInputAutoCompleteDone(b) – 'b' is true to signify that the autocomplete process is done.

setUnitInputUsePlurality(b) – 'b' is true to use plurals.

setUnitInputAutoComplete(b) – 'b' is true to allow autocomplete.

setWholeNumberInputLabel(data) – 'data' is a string for the label for the whole number input textbox.

setUnitInputLabel(data) – 'data' is a string for the label for the units input textbox.

ScrollPane

The ScrollPane widget is a container on which to place other widgets and allows vertical scrolling.

addObject(object, column) – Adds a widget to the scrollpane. 'object' is the widget to add. 'column' should be 1 or not provided.

scrollIntoView(objectName, pct) – Scrolls provided object into view. If 'objectName' is "TOP", then the scrollpane is scrolled to the top. If 'objectName' is "END", then the scrollpane is scrolled to the bottom. Otherwise, the scrollpane is scrolled to the object provided. 'pct' is the percentage of the viewport where the object should appear, between 0 and 1.

setZOrder(objectName, idx) – Sets the z-order for the object for painting. All other objects will shift their z-order accordingly. 'objectName' is the name of the object. 'idx' is the z-order index.

getZOrder(objectName, idx) – Returns the z-order index for the provided object.

Footer

The Footer widget displays information at the bottom of the frame. Use of this widget reduces the vertical size of the page view.

addObject(object) – Adds a widget to the footer. 'object' is the widget to add.

setProgressBar(pct) – Sets the progress bar within the footer. 'pct' is the percent complete from 0 to 1.

setProgressBarVisible(b) – 'b' is true to show the footer progress bar.

DialogBox

The DialogBox widget displays text in a rectangle along with a button for user action.

setText(text, mode) – Sets the text to display in the dialog box. The 'text' is a paragraph, and 'mode' is the paragraph mode.

addButton(pageID, func) – Creates a button for the provided 'pageID', and calls 'func' when the button is pressed.

Tooltip

The Tooltip widget displays text within a rectangle.

setText(text, mode) – Sets the text to display in the dialog box. The 'text' is a paragraph, and 'mode' is the paragraph mode.

setPointerPosition(pctxPointer, useTooltipWidth) – 'pctxPointer' is the percentage of the pane, or, if 'useTooltipWidth' = true, then pctxPointer is the percentage of the tooltip width.

setFillColor(color) – 'color' is an array of red, green, blue values.

setPointerDirection(direction) – Sets the direction of the tooltip pointer. One of self.pointer.directionIDs.TOP, self.pointer.directionIDs.RIGHT, or self.pointer.directionIDs.BOTTOM.

setPointerHeight(nonScaledHeight) – Sets height of the pointer in pixels.

Image

The Image widget displays a graphic.

loadImage(resourceImage) – Creates a new instance of an image. 'resourceImage' is in the format of an nSpire resource.

rotate(angle) – Rotates the image by 'angle' degrees from current angle.

rotateFromZeroDeg(rotation) – Rotates the image by 'rotation' degrees from 0 degrees.

Depiction

The Depiction widget displays a graphic or line drawing representation of the graphic.

rotate(angle) – Rotates the image by 'angle' degrees from current angle.

rotateFromZeroDeg(rotation) – Rotates the image by 'rotation' degrees from 0 degrees.

setFillColor(color) – 'color' is an array of red, green, blue values.

setBorderColor(color) – 'color' is an array of red, green, blue values.

setFigure(figureType) – Sets the figure type to one of app.model.figureTypeIDs: "RIGHT_ARROWHEAD", "LEFT_ARROWHEAD", "DOWN_ARROWHEAD", "UP_ARROWHEAD", "CHECK_MARK", "TICK_MARK", "SMALL_RIGHT_ARROW", "SMALL_LEFT_ARROW", "SMALL_DOWN_ARROW", "SMALL_UP_ARROW", "CIRCLE", "TRIANGLE", "STAR", "DIAMOND", "SQUARE", "UP_ARROW", "DOWN_ARROW", "LEFT_ARROW", "RIGHT_ARROW", "HOUR_GLASS"

setDrawStyle(style) – 'style' is one of app.graphicsUtilities.drawStyles.OUTLINE_ONLY, app.graphicsUtilities.drawStyles.FILL_AND_OUTLINE, or app.graphicsUtilities.drawStyles.FILL_ONLY

loadImage(resourceImage) – Creates a new instance of an image. 'resourceImage' is in the format of an nSpire resource.

Figure

The Figure widget displays a line drawing. The following is a list of widgets that are inherited from Figure:

Circle, Rectangle, RoundedRectangle, OctagonRectangle, Star, FigureCheckMark, FigureCrossMark, FigureRightArrow, FigureLeftArrow, FigureDownArrow, FigureUpArrow, FigureDiamond, FigureSquare, FigureTriangle, CompoundFigure, HourGlass

setFillColor(color) – 'color' is an array of red, green, blue values.

setBorderColor(color) – 'color' is an array of red, green, blue values.

setDrawStyle(style) – 'style' is one of app.graphicsUtilities.drawStyles.OUTLINE_ONLY, app.graphicsUtilities.drawStyles.FILL_AND_OUTLINE, or app.graphicsUtilities.drawStyles.FILL_ONLY

setLocalScale(w, h) – Sizes the figure with a bounding box of 'w' width, and 'h' height in pixels.

Symbol

The Symbol widget displays either FigureDiamond, FigureSquare, FigureTriangle, or Circle depending on the style.

setStyle(style) – 'style' is one of "FilledDiamond", "FilledCircle", "FilledSquare", "FilledTriangle", "EmptyDiamond", "EmptyCircle", "EmptySquare", or "EmptyTriangle".

setLocalScaleDiamond(w, h) – Sizes the diamond figure with a bounding box of 'w' width, and 'h' height in pixels.

setLocalScaleSquare(w, h) – Sizes the square figure with a bounding box of 'w' width, and 'h' height in pixels.

setLocalScaleTriangle(w, h) – Sizes the triangle figure with a bounding box of 'w' width, and 'h' height in pixels.

Feedback

The Feedback widget displays a checkmark or crossmark.

setStyle(style) – 'style' is one of "CrossMark" or "CheckMark".

setLocalScaleCrossMark(w, h) – Sizes the crossmark with a bounding box of 'w' width, and 'h' height in pixels.

setLocalCheckMark(w, h) – Sizes the checkmark with a bounding box of 'w' width, and 'h' height in pixels.

9. User Interface Manager

The User Interface Manager manages the portion of the user interface that includes tab stops, keyboard focus, and a hierarchy of events. The UI Manager is implemented via the UIMgr class in the framework. The following are the UI Manager functions and properties that the programmer may use.

app.frame.UIMgr:addPageTabObjects(pageID, list) – The programmer calls this function to tell the UIMgr which widget objects should be included in the tab stop list. The pageID parameter is used to identify the view page containing the UI objects, since each view page has its own list of UI objects and tab stops. For example, calling app.frame.UIMgr:addPageTabObjects(self.pageID, {app.frame.buttonMenu, self.myButton}) will add the frame's buttonMenu and the view's myButton to the UIMgr for the view page making the call.

app.frame.UIMgr:setPageTabSequence(pageID, list) – This function looks very similar to addPageTabObjects(), but the only action it performs is to set the tab order using the order of the objects found list 'list'.

app.frame.UIMgr:switchFocus(name) – The programmer calls this function to set the keyboard focus to a particular widget. 'name' is the unique identifier of the widget. When the user uses the tab

or back-tab key, the UIMgr will automatically switch the focus to the next widget in the tab sequence and remove the focus from the previous widget. The widget provides code in paint() to identify to the user that the widget has focus, for example with a red rectangle.

app.frame.UIMgr:deactivateObjects(viewID) – This function deactivates all UI Manager objects so that the user cannot interact with any object.

app.frame.UIMgr:activateObjects(viewID) – This function reactivates all UI Manager objects to their previous state. To be reactivated, the object must previously have been visible.

app.frame.UIMgr:addListener(viewID, listener) – This function adds a listener function to the UI Manager. 'viewID' identifies the view that is listening. 'listener' is the function that receives notification when a UI Manager event occurs. Return is a listenerID that the view should store for reference.

The UI Manager will notify listeners for the following events: EVENT_MOUSE_DOWN, EVENT_MOUSE_MOVE, EVENT_MOUSE_OUT, EVENT_MOUSE_OVER

app.frame.UIMgr:removeListener(viewID, listenerID) – This function removes an existing listener. 'viewID' identifies the view that contains the listener. 'listenerID' is the ID of the listener that was returned via addListener().

app.frame.UIMgr:getObjectByName(objectName) – This function returns the object when provided the string name of the object.

app.frame.UIMgr:enterKey() – The programmer calls this function to allow the UI Manager to handle the enter key. The UI Manager will call the enterKey() function for the widget that has focus and allow the widget to handle the enterKey. An example might be the widget switching focus to a different part of the widget. If the widget handled the enter key, the return is true, otherwise the return is false.

app.frame.UIMgr:hasCollided(obj1, obj2, sidesOffset, areaExtension) – This function determines if two objects have collided. 'obj1' is the first object. 'obj2' is the second object. 'sidesOffset' is an amount in pixels to increase the area being detected. 'areaExtension'

app.frame.UIMgr.currentObject – This property contains the object that has keyboard focus.

app.frame.UIMgr.activeViewID – This property contains the active view ID.

app.frame.UIMgr.tabSequence[viewID][objectName] – This property is a double array that contains the tab sequence. 'viewID' is the index for the view ID. 'objectName' is the string name of the UI object.

The example below shows how to set the tab stops for the main menu button and a button object called button1 within setupView. It also shows how to set the initial focus to the menu button during stateInit.

```
1.  function MainPageView:setupView()
2.      local list = {app.frame.buttonMenu, self.button1}
3.      app.frame.UIMgr:addPageTabObjects(self.pageID, list)
4.      app.frame.UIMgr:setPageTabSequence(self.pageID, list)
```

```
5.  end
6.
7.  function MainPageView:stateInit()
8.      app.frame.UIMgr:switchFocus("frame_buttonmenu")
9.  end
```

10. Menu

The menu is a built-in widget that has two levels and will typically be populated with the same menu and submenu items as the TI-nspire menu. The menu is attached to the menu button which is contained within the window header. To populate the menu and submenu items, the programmer first calls app.model:setMenuIDs() and app.model:setSubMenuIDs() within ModuleModel:configure() to create IDs associated with the menu. Then, the programmer calls setMenu() to create the menu hierarchy. The programmer may enable the menu button and menu by calling app.model:setShowMenuButton(true) within ModuleModel :configure(). An example is shown below.

```
1.  function ModuleModel:configure()
2.      app.model:setMenuIDs({"PRACTICE"})
3.      app.model:setSubMenuIDs({"ADDITION"})
4.      app.model:setMenu({"Practice", {"Addition", function()
    app.controller.moduleController:handleMenu(app.model.
    menuChoicesList.PRACTICE, app.model.subMenuChoicesList.ADDITION
    ) end), },})
5.      app.model:setShowMenuButton(true)
6.  end
7.
8.  function ModuleController:handleMenu(menu, submenu)
9.      --handle menu item
10. end
```

The framework contains a feature that displays an animated yellow arrow that points to the menu button. The yellow arrow will appear at the start of the application to inform the user how to begin. The default wait time before displaying the yellow arrow is 10 seconds, but this time can be set by calling app.model.setMenuArrowWaitTime(t), where 't' is the wait time in milliseconds. The programmer may call app.frame:disableMenuArrow() to disable the animated yellow arrow.

11. Soft Keyboard

The soft keyboard is a framework widget that displays an on-screen keyboard for situations where no physical keyboard is available, such as touch screen devices. To configure the soft keyboard, the programmer creates a class called KeyboardLayout. The soft keyboard is accessed via app.frame.keyboard.

KeyboardLayout has the following properties.

keys[idx] - This array property contains the array of keys to display. Since more than one keyboard layout can exist, 'idx' is a reference to which layout is being referenced. Within the keys property is a five-element array. Element 1 contains the string to display on the key. Elements 2 and 3 contain the x and y percentages within the keyboard pane for positioning the key. Elements 4 and 5 contain the x and y pixel sizes of the key.

frameSize[idx] – This property contains the x and y pixel sizes of the keyboard pane.

grabIconXY[idx] - This property contains the x and y pixel location of the grab handle for dragging the keyboard.

framePctPositions[idx] – This property contains the x and y percentages for location within the keyboard container pane.

An example is shown below.

```
1.  KeyboardLayout = class()
2.
3.  function KeyboardLayout:init()
4.      self.keys, self.frameSize, self.framePctPositions,
    self.grabIconXY = {}, {}, {}, {}
5.      local idx = 1
6.
7.      self.keys[idx] = { {"1", 0, 0, 25, 25}, {"2", 0, 0, 25,
    25},{"Enter", .79, .5, 41, 25}, }
8.      self.frameSize[idx] = {232, 60}
9.      self.grabIconXY[idx] = {-12, 39}
10.     self.framePctPositions[idx] = {.02, .65}
11. end
```

The soft keyboard will display automatically on touch screen devices for input widgets. To force the soft keyboard to display, the programmer can call app.model:setKeyboardOn(true). Figure 12 shows an example of a soft keyboard layout.

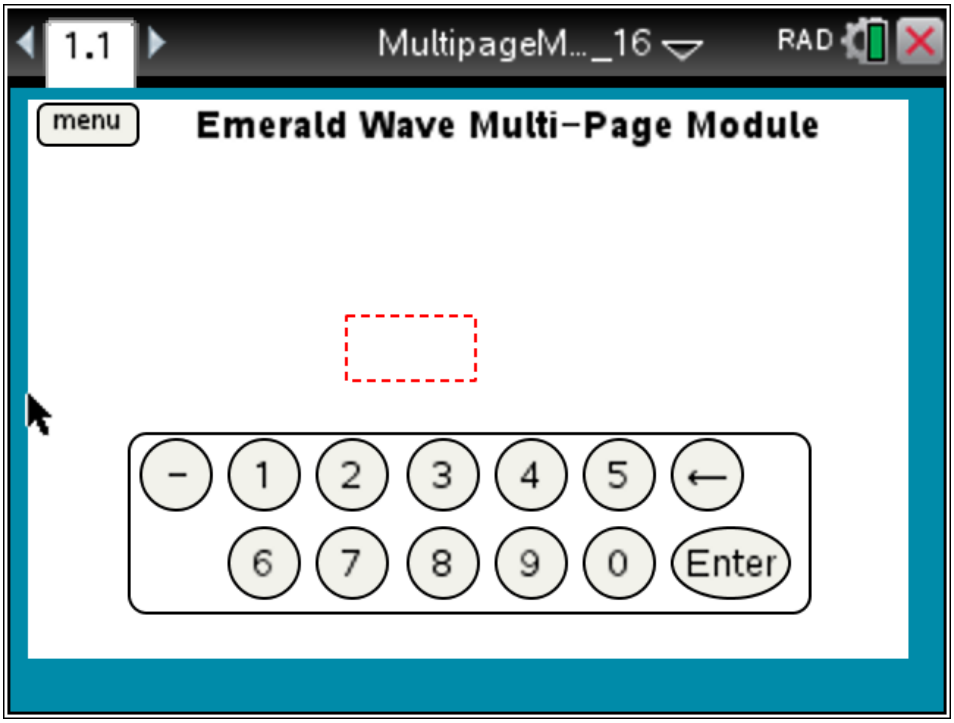

Figure 12 - Soft Keyboard

12. Event Flow

The framework startup sequence is based on the event model of the TI-nspire API. During startup, the programmer is given opportunity by the framework to define pages and views, create user interface objects and define program flow. Figure 13 shows the first step in the process, on.construction().

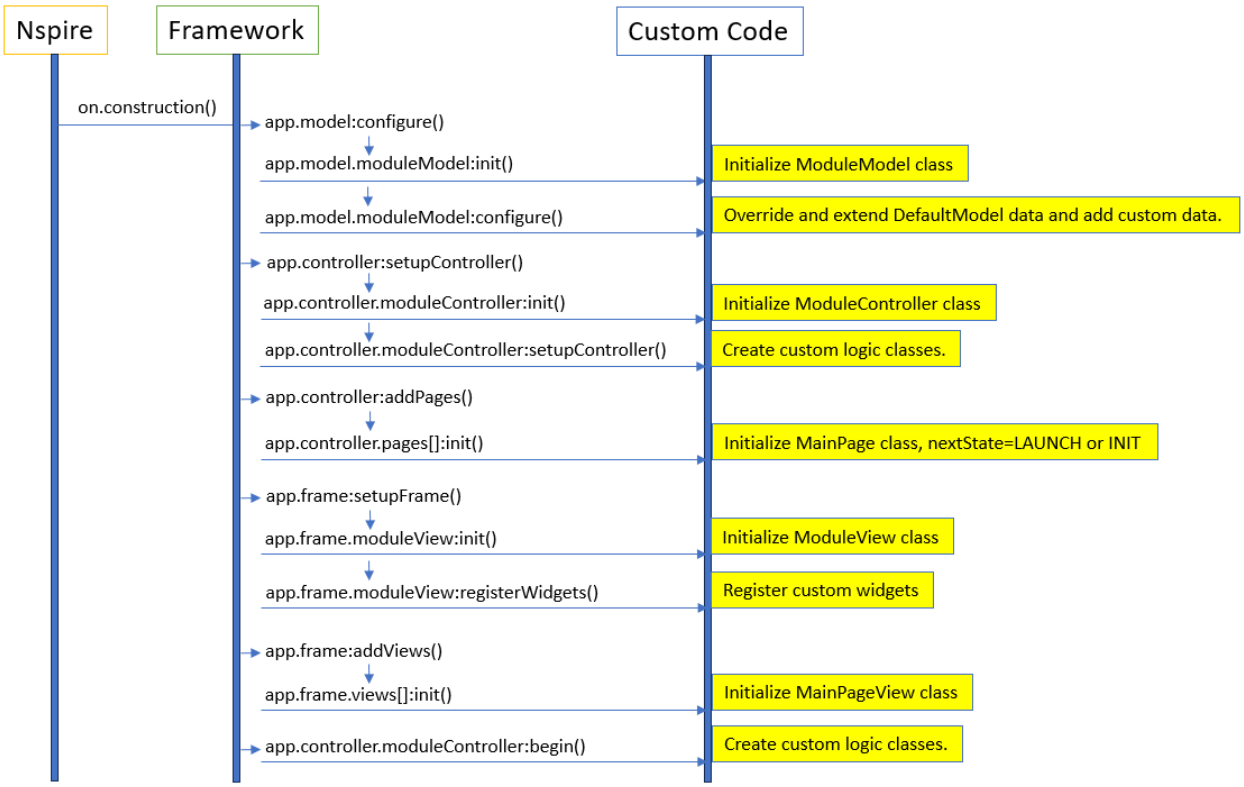

Figure 13 - Event Flow - on.construction()

on.onstruction() is the first event in the TI-nspire API. The EW framework is constructed during this event, including the creation of the Model, Frame, and Controller classes. During on.construction(), there are three methods that can be called by the framework to allow for customization of the application by the programmer:

ModuleModel:configure() – The framework contains a built-in class called DefaultModel which provides basic data structures which allow the entire framework to run without modification. The default data is accessed via the app.model table. The programmer may provide a class called ModuleModel which contains a method called configure(). During on.construction(), app.model.moduleModel:configure() will be called which allows the programmer to override default data values, extend existing tables, or create new data structures. This custom data can be added directly to the app.model table and accessed via app.model.

ModuleController:setupController() – The programmer may create a class called ModuleController which contains a method called setupController(). If this class and function exist, the framework will call app.controller.moduleController:setupController() which allows the programmer to create custom logic classes that will control program flow.

ModuleView:registerWidgets() – The programmer may create a class called ModuleView which contains a method called registerWidgets(). If this class and function exist, the framework will call app.frame.moduleView:registerWidgets() which allows the programmer to register custom widgets.

ModuleController:begin() – Once the framework has been constructed, the last method call from on.construction() is to app.controller.moduleController:begin() which is the method where custom global data may be populated.

After on.construction(), the data structures have been created, and the TI-nspire API will call on.resize(w, h) with a width and height of the main window. Figure 14 shows the framework methods that are invoked during on.resize().

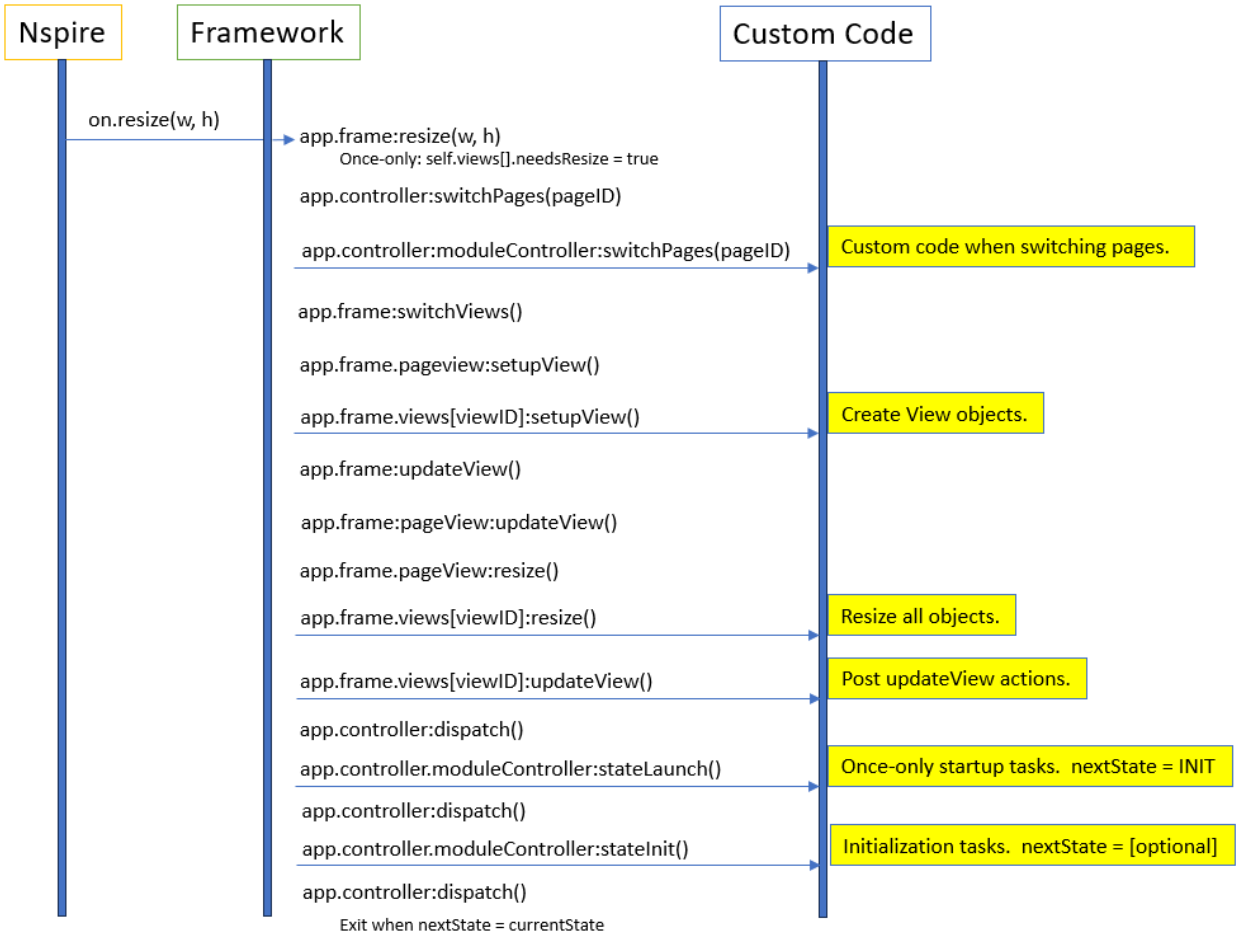

Figure 14 - Event Flow – on.resize()

It is important to note that the very first time through on.resize() is different than all other subsequent calls to on.resize(). During the first call, the variable app.frame.needsResize will be nil, which will indicate to the framework that this is the first call to on.resize(). The framework will then call the startup page, specified by app.model.startupPageID, and then call switchPages().

ModuleController:switchPages() – The framework calls app.controller.moduleController:switchPages() at the beginning of the page switch process. The programmer may use this awareness of the upcoming page switch and take any necessary actions.

view:setupView() – During the call to app.frame:switchViews(), a call is made to the page view, which was created during on.construction(). The programmer should use this opportunity to create user interface objects during this call and may set a flag so that the setup only occurs once for the lifetime of the app.

view:resize() – During the call to app.frame:updateViews(), a call is made to resize the view. The programmer can scale and resize all UI objects during this call.

view:updateView() – After the framework calls updateView() to complete the resize process, a call is made to view:updateView() to allow the programmer to make take necessary post-update actions.

ModuleController:stateLaunch() – Logic flow is configured by states, with the dispatcher calling each state in turn. A built-in state called stateLaunch is intended to be called and used only once during the lifetime of the app. The programmer can take once-only actions during the call to app.controller.moduleController:stateLaunch().

ModuleController:stateInit() – stateInit is another built-in state that is intended to be used as a reset, setting up the view into its initial state, such as when switching pages, during the call to app.controller.moduleController:stateInit().

[customState]() – The programmer may define additional custom states that the dispatcher can call. The programmer can control the logic flow within a page by using these custom states.

Once the app is constructed and the view is setup and resized, the final TI-nspire API call is to on.paint(), as shown in Figure 15.

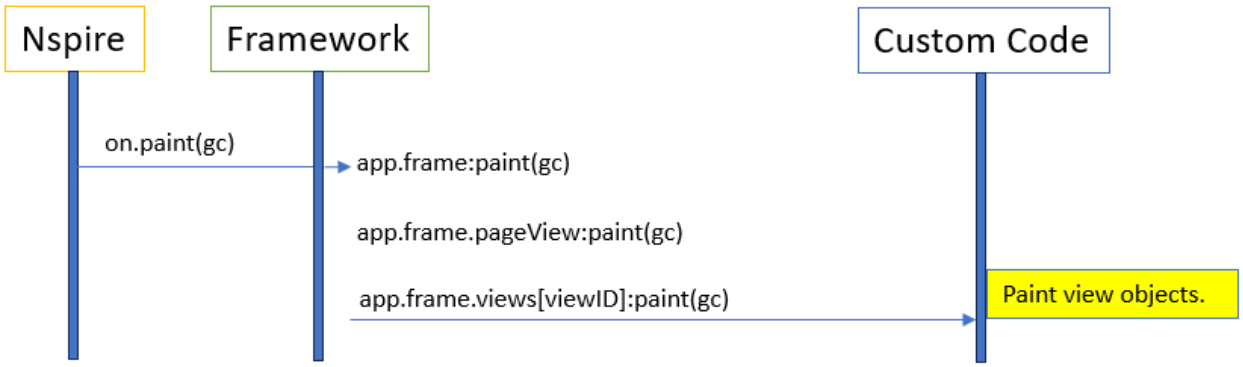

Figure 15 - Event Flow - on.paint()

The programmer will typically use widgets that sit on a page view and will not use on.paint() directly. However, the option to handle on.paint() within a page view is available to allow the programmer to draw directly onto the view.

13. DefaultModel

The DefaultModel class contains data for the application and is implemented as the app.model table. Wrapper functions offer access to some of the default data, allowing the programmer to modify the default data**.** These wrapper functions are accessed via app.model.

setPageIDs(list) – Creates an enumerated list of page IDs. This function overrides any existing list.

setPageList(list) – Creates a list of pages consisting of a logic class, view class, and name.

setStatesList(list) – Creates a list of program flow states.

setSubStatesList(list) – Creates a list of program flow substates.

setStateFunctions(list) – Sets the function to call for each defined state.

setNextSubStateTable(table) – Creates a sequenced list of substates.

setAllowedUserInputCatagories(list) – Creates a list of user input categories that can be used to enable or disable user input. The Frame class uses the item “ANIMATION”, and the Controller class uses the item “POST_MESSAGE”.

setStartupPageID(pageID) – Sets the startup page ID.

setPageBorderSize(size) – Sets the Frame border size. The view of the page lies within the border.

setBackgroundColor(color) – Sets a Frame background color.

setFrameBorder(b) – Turns the Frame border on or off.

setFrameBorderColor(color) – Sets a Frame border color.

setFrameDivider(b) – Turns the top divider line between the frame and the view on or off.

setMenuArrowWaitTime(t) – Sets the length of time for the menu arrow to animate.

setMenuArrowEnabled(b) – Enables or disables the menu arrow.

setShowMenuButton(b) – Enables or disables the Menu button.

setShowViewButton(b) – Enables or disables the Show View button.

setShowHeader(b) – Turns the Frame header on or off.

setShowFooter(b) – Turns the Frame footer on or off.

setHasKeyboard(b) – Enables or disables the automatic soft keyboard.

setKeyboardInputs(list) – Defines a list of Widgets that accept keyboard input.

setKeyboardOn(b) – Makes the soft keyboard visible or hidden.

registerTimerIDs(names) – Registers a list of custom timer IDs. The Default timer IDs are "POSTMESSAGETIMER", "FRAMETIMER", "MENUTIMER", and "MENUARROWTIMER". Registration adds to the existing default list.

registerTimerFunctions(items) – Registers a list of additional timer handlers for custom timers.

registerEvents(names) – Registers a list of custom events. Default events are "EVENT_NONE", "EVENT_MOUSE_MOVE", "EVENT_MOUSE_DOWN", "EVENT_MOUSE_UP", "EVENT_MOUSE_OUT", "EVENT_MOUSE_OVER", "EVENT_ARROW_LEFT", "EVENT_ARROW_RIGHT", "EVENT_PARAGRAPH_ANIMATION_END", "EVENT_CIRCLE_ANIMATION_END", "EVENT_HIGHLIGHT_ANIMATION_END", "EVENT_STADIUM_ANIMATION_END", "EVENT_LOST_FOCUS", "EVENT_GOT_FOCUS", "EVENT_SCROLL", "EVENT_BUTTON_NEXT", "EVENT_BUTTON_1", "EVENT_POPUP_KEYBOARD", "EVENT_SIZE_CHANGE", "EVENT_ENTER_KEY", "EVENT_CHAR_IN", "EVENT_BACKSPACE". Registration adds to the existing default list.

registerScenes(names) – Registers a list of movie scenes. Default scenes are "SCENE_START", "SCENE_1", "SCENE_2", "SCENE_3", "SCENE_4", "SCENE_5", "SCENE_6", "SCENE_7", "SCENE_8", "SCENE_END". Registration adds to the existing default list.

setMenuIDs(list) – Sets a list of menu IDs.

setSubMenuIDs(list) – Sets a list of submenu IDs.

setMenu(list) – Sets the menu and submenu items.

setPageMaxLoadTimeDesktop(t) – Sets a load time maximum for a desktop device. If the application load time exceeds this amount, the device will be considered slow.

setPageMaxLoadTimeWeb(t) - Sets a load time maximum for when the application is running within a browser. If the application load time exceeds this amount, the device will be considered slow.

setPlatformIsSlow(b) – Forces the application to consider the device as slow or normal.

14. ModuleModel

The ModuleModel class is an optional class provided by the programmer. If supplied by the programmer, the method app.model.moduleModel:configure() is called during app.model:configure() within on.construction() and after the creation of the default data. Within app.model.moduleModel:configure(), the programmer may make calls to the default model wrapper functions to modify default data and add or modify app.model tables directly.

Identifiers for each page and view are created during the call to app.model.moduleModel:configure(). The example below shows how to call the app.model wrapper functions to set the pages, views, menu, and frame properties.

```
1.  function ModuleModel:configure()
2.      app.model:setPageIDs({"MAIN_PAGE"})
3.      app.model:setPageList({[app.model.pageIDs.MAIN_PAGE] =
    {MainPage, MainPageView, "MAIN"}})
4.     app.model:setMenuIDs({"MAIN_MENU"})
5.     app.model:setSubMenuIDs({"SUB_MENU"})
6.     app.model:setMenu({{"Main Menu", {"Sub Menu", function()
    app.controller.moduleController:handleMenu(1, 1) end}, }, })
```

```
7.     app.model:setFrameBorder(true)
8.  end
```

app.model.moduleModule:configure() is the only method in the ModuleModel class that is called by the framework, and this method is only called once, which is during the TI-nspire on.contruction() event. The programmer may add custom methods to the ModuleModel class that relates to application data.

15. ModuleView

The ModuleView class is an optional class provided by the programmer. If supplied by the programmer, the app.frame.moduleView instance is created when the framework creates the frame. **registerWidgets()** – This method is called by the framework only once, which is during the TI-nspire on.contruction() event. During the creation of the frame, and if supplied by the programmer, app.frame.moduleView:registerWidets() is called to register new widgets supplied by the programmer. The example below shows how to register a custom widget called SimpleNumberLine.

```
1.  function ModuleView:registerWidgets()
2.     app.frame.widgetMgr:register("NUMBER_LINE", function(name,
    options) local widget = SimpleNumberLine(name);
    app.frame.widgetMgr:addWidget(widget, options) return widget
    end)
3.  end
```

handleUserAction(view) – This method, if supplied by the programmer, is called by the framework for any user action, which thus allows the programmer to apply a global action for all user actions. 'view' is the view receiving the user action.

arrowLeft(view) – The programmer may handle an arrowLeft keypress for the entire view. The return should be true if the framework should take no further action for the arrowLeft keypress.

arrowRight(view) – The programmer may handle an arrowRight keypress for the entire view. The return should be true if the framework should take no further action for the arrowRight keypress.

arrowUp(view) – The programmer may handle an arrowUp keypress for the entire view. The return should be true if the framework should take no further action for the arrowUp keypress.

arrowDown(view) – The programmer may handle an arrowDown keypress for the entire view. The return should be true if the framework should take no further action for the arrowDown keypress.

mouseDown(view) – The programmer may handle a mouseDown for the entire view. The return should be true if the framework should take no further action for the mouseDown event.

mouseUp(view) – The programmer may handle a mouseUp event for the entire view. The return should be true if the framework should take no further action for the mouseUp event.

mouseMove(view) – The programmer may handle a mouseMove event for the entire view. The return should be true if the framework should take no further action for the mouseMove event.

grabUp(view) – The programmer may handle a grabUp event for the entire view. The return should be true if the framework should take no further action for the grabUp event.

releaseGrab(view) – The programmer may handle a releaseGrab for the entire view. The return should be true if the framework should take no further action for the releaseGrab event.

charIn(view) – The programmer may handle a character keypress for the entire view. The return should be true if the framework should take no further action for the character keypress.

backspaceKey(view) – The programmer may handle a backspace keypress for the entire view. The return should be true if the framework should take no further action for the backspace keypress.

tabKey(view) – The programmer may handle a Tab keypress for the entire view. The return should be true if the framework should take no further action for the Tab keypress.

backTabKey(view) – The programmer may handle a Back Tab keypress for the entire view. The return should be true if the framework should take no further action for the Back Tab keypress.

shiftArrowLeft(view) – The programmer may handle a shift+arrowLeft keypress for the entire view. The return should be true if the framework should take no further action for the shift+arrowLeft keypress.

shiftArrowRight(view) – The programmer may handle a shift+arrowRight keypress for the entire view. The return should be true if the framework should take no further action for the shift+arrowRight keypress.

shiftArrowUp(view) – The programmer may handle a shift+arrowUp keypress for the entire view. The return should be true if the framework should take no further action for the shift+arrowUp keypress.

shiftArrowDown(view) – The programmer may handle a shift+arrowDown keypress for the entire view. The return should be true if the framework should take no further action for the shift+arrowDown keypress.

homeKey(view) – The programmer may handle a Home keypress for the entire view. The return should be true if the framework should take no further action for the Home keypress.

endKey(view) – The programmer may handle an End keypress for the entire view. The return should be true if the framework should take no further action for the End keypress.

deleteKey(view) – The programmer may handle a Delete keypress for the entire view. The return should be true if the framework should take no further action for the Delete keypress.

enterKey(view) – The programmer may handle an Enter keypress for the entire view. The return should be true if the framework should take no further action for the Enter keypress.

selectKey(view) – The programmer may handle a Select keypress for the entire view. If there is no method to handle the selectKey in ModuleView class, then the framework will call the enterKey()

method instead, if it exists. The return should be true if the framework should take no further action for the Enter keypress.

escapeKey(view) – The programmer may handle an Esc keypress for the entire view. The return should be true if the framework should take no further action for the Esc keypress.

cut(view) – The programmer may handle a Cut event for the entire view. The return should be true if the framework should take no further action for the Cut Event.

copy(view) – The programmer may handle a Copy event for the entire view. The return should be true if the framework should take no further action for the Copy Event.

paste(view) – The programmer may handle a Paste event for the entire view. The return should be true if the framework should take no further action for the Paste Event.

The programmer may add custom methods to the ModuleView class that relate to the view and user interface for the entire application.

16. PageView

PageView class is a framework class. The programmer can access methods in the PageView class via the app.frame.pageView table. The PageView class handles layout duties and event processing for a page view. While most of the functions are used only by the framework, the following methods are available to the programmer.

initPageView(view, name) – The programmer should call this function within the init() function to initialize variables. 'view' is the view being initialized. 'name' is a unique string name for the view. This call will initialize view.needsViewSetup = true so that the PageView class knows that the user interface objects need to be set up.

setupView(view) – The programmer should call app.frame.updateView(view) to setup the user interface objects. This call causes the framework to call the [view]:setupView() method if view.needsViewSetup is true. The programmer should assign self.needsViewSetup = false at the completion of the setup.

updateView(view) – The programmer should call app.frame.updateView(view) at any point where the layout needs updating, such as at the end of setting up an initial state.

An example is shown below.

```
1.  MainPageView = class()
2.
3.  function MainPageView:init(name)
4.      app.frame.pageView:initPageView(self, name)
5.  End
6.
7.  function MainPageView:stateInit()
8.     app.frame.pageView:setupView(self)
9.     self.needsViewSetup = false
10.    app.frame.pageView:updateView(self)
```

```
11. end
```

drawCheckMark(gc, x, y, w, h, scale) – This method can be used as the callback for drawing a checkmark, such as on a button. 'gc' is the graphics context. x,y are the pixel coordinates. w, h are the width and height. scale is the current drawing scale for the view.

setObjectVisibility(visibleTable, hideTable, activeTable, inactiveTable) – This is a convenience method that allows the programmer to set the visible and active states of all UI widgets at one time, such as during stateInit. 'visibleTable' is an array of UI objects and the .visible property will be set to true. 'hideTable' is an array of UI objects and the .visible property will be set to false. 'activeTable' is an array of UI objects and the .active property will be set to true. 'inactiveTable' is an array of UI objects and the .active property will be set to false.

setMovieView(view) – Sets the app.frame.pageView.movieView property to the provided 'view' so that app.frame.pageView is aware of which view is playing a movie.

playMovie(event, scene) – Causes app.frame.pageView to step through a set of steps to play an animation or "movie", which includes events and scenes. 'event' is one of the events registered via app.model:registerEvents(). 'scene' is a property of app.model.scenes: SCENE_START, SCENE_1, SCENE_2, SCENE_3, SCENE_4, SCENE_5, SCENE_6, SCENE_7, SCENE_8, SCENE_END, or a custom scene provided via app.model:registerScenes().

setupImages(view) – The programmer may call this function to setup the wait cursor by supplying the view.

attachKeyboard(view) – Attaches the soft keyboard to the view.

detachKeyboard(view) – Detaches the soft keyboard to the view.

17. View

The frame can contain multiple views. Each view, which is for the user interface, is associated one-to-one with a page, which is for logic flow. Figure 16 shows the frame holding a view. The view displays in the space between the header and the footer. An optional scroll pane can be added to the view, as shown in Figure 16. The text, input boxes, buttons, and tooltip are user interface widgets that can be optionally added to the view or the scroll pane. Figure 16 also shows a +/- button, which is part of the header of the frame and allows the user to zoom in or out on devices with larger screens.

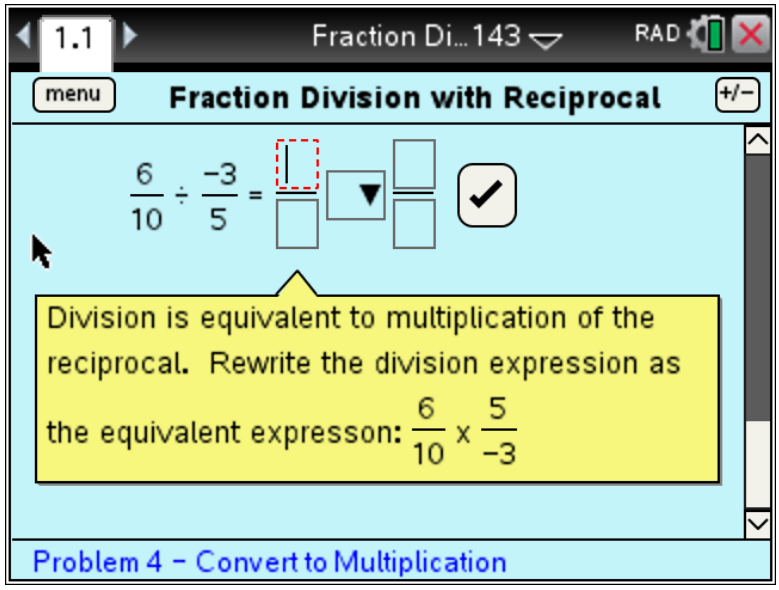

Figure 16 - View with Scroll Pane

A basic view class called MainPageView is shown in the example below.

```
1.  MainPageView = class()
2.
3.  function MainPageView:init(name)
4.      app.frame.pageView:initPageView(self, name)  --initialize
    variables.
5.  end
6.
7.  function MainPageView:setupView()
8.      -- create widgets, add tabs, set initial focus
9.  end
10.
11. function MainPageView:layoutY()
12.     -- layout the y positioning of widgets
13. end
14.
15. function MainPageView:layoutXOneColumn()
16.     -- layout the x positioning of widgets
17. end
18.
19. function MainPageView:stateInit()
20.     -- Set initial visibility of objects, initial focus, and
    scroll pane positioning.
21.     app.frame.pageView:updateView(self) --Reposition scrollpane
    objects and update the virtual height.
22. end
```

18. Controller

The Controller class in the framework is implemented as the app.controller table, controls logic flow for the framework, and includes a state machine dispatcher. The programmer may call the following functions within the Controller class.

dispatch(pageID) – The programmer may need to call the dispatcher directly, for example in response to a user action. The programmer calls app.controller:dispatch(pageID), where pageID is typically the current page. The dispatcher will then check the nextState and nextSubState properties of the provided pageID and will call the appropriate function. The dispatcher will loop until both the nextState and nextSubState are unchanged after the function call. Upon exiting the loop, the code will return to the line immediately after the call to dispatch().

postMessage(msg) – The programmer may call postMessage to add a message to the message queue. The framework will then start the timer called POSTMESSAGETIMER. The framework will also disable all user input at this time. After the message is posted to the queue, the code will return to the next line immediately after the call to postMessage() and continue. During the next POSTMESSAGETIMER timer tick, the framework will remove the message from the queue and execute the message as a function.

The purpose of postMessage() is to allow the code to temporarily return to the operating system for actions such as paint(). An example of using postMessage and dispatch to provide an opportunity for the operating system to paint the clicked button is shown below.

```
1.  function MainPage:gotoNextState()
2.
3.     app.frame.imageWaitCursor:setVisible(true)
4.
5.     app.controller:postMessage({function(...)
6.        self.nextState = nextState
7.        self.nextSubState = nextSubState
8.        app.controller:dispatch(self.pageID)
9.     end})
10. end
```

19. ModuleController

The ModuleController class is an optional class provided by the programmer to allow for custom program control. If supplied by the programmer, the app.controller.moduleController table is created during on.contstruction().

The figure below shows how to create a custom ModuleController class that handles the states LAUNCH, INIT, and END. It also shows how to handle moving the dispatcher from one substate to the next substate.

```
1.  ModuleController = class()
2.
3.  function ModuleController:init() end
```

```
4.
5.  function ModuleController:stateLaunch(page) if page.stateLaunch
    then page:stateLaunch() end end
6.  function ModuleController:stateInit(page) if page.stateInit
    then page:stateInit() end end
7.  function ModuleController:stateEnd(page) if page.stateEnd then
    page:stateEnd() end end
8.
9.  function ModuleController:nextSubState(page)
10.     local nextSubState =
    app.model.nextSubStateTable[page.subState]
11.
12.     if nextSubState ~= nil then page.nextSubState =
    nextSubState end
13.
14.     app.controller:dispatch(page.pageID)
15. end
16.
17. function ModuleController:handleMenu(menuID, submenuID)
18.
```

20. Page

The Controller can contain multiple logic pages.

```
1.  ModuleController = class()
2.
3.  function ModuleController:init() end
4.
5.  function ModuleController:stateLaunch(page) if page.stateLaunch
    then page:stateLaunch() end end
6.  function ModuleController:stateInit(page) if page.stateInit
    then page:stateInit() end end
7.  function ModuleController:stateEnd(page) if page.stateEnd then
    page:stateEnd() end end
8.
9.  function ModuleController:nextSubState(page)
10.     local nextSubState =
    app.model.nextSubStateTable[page.subState]
11.
12.     if nextSubState ~= nil then page.nextSubState =
    nextSubState end
13.
14.     app.controller:dispatch(page.pageID)
15. end
16.
17. function ModuleModel:configure()
18.     model:setStatesList({"INIT", "MAIN", "END"})
```

```
19.     model:setSubStatesList({ "SETUP", "USER_INPUT",
        "PROCESS_INPUT"})
20.     model:setNextSubStateTable({[app.model.subStatesList.SETUP]
        = app.model.subStatesList.USER_INPUT,
        [app.model.subStatesList.USER_INPUT] =
        app.model.subStatesList.PROCESS_INPUT,})
21. end
22.
23. function MyPage:stateLaunch()
24.     --handle one-time launch logic
25.     self.view:stateLaunch()    --one-time view actions
26. end
27.
28. function MyPage:stateInit()
29.     --handle reset and initialization logic.
30.     self.view:stateInit()    --reset and initialize the view.
31. end
32.
33. function MyPage:stateEnd()
34.     --handle end state logic.
35.     self.view:stateEnd()   --take any necessary view actions.
36. end
```

21. User Input Events

The framework provides the following keyboard, mouse, and touch pad events, all of which are forwarded to app.frame.pageView.moduleView for possible handling by the programmer.

handleUserAction(view) – Each of the events listed below, with the exception of mouseUp(), will first call app.frame.moduleView:handleUserAction() to allow the programmer to take a global action prior to handling the individual event.

Each event listed below is an optional method of app.frame.moduleView supplied by the programmer.

arrowLeft(view) – Called when the user presses the left arrow key.

arrowRight(view) – Called when the user presses the right arrow key.

arrowUp(view) – Called when the user presses the up arrow key.

arrowDown(view) – Called when the user presses the down arrow key.

mouseDown(view, x, y) – Called when the user presses the mouse down button.

mouseUp(view, x, y) – Called when the user releases the mouse down button.

mouseMove(view, x, y) – Called when the user moves the mouse.

grabUp(view, x, y) – Called when the user releases the grab button on the nspire. There is no framework event for grabDown.

releaseGrab(view) - Whenever there is user input, releaseGrab() will be called to allow the programmer to release an existing grabbed object.

charIn(view, char) - Called when the user presses a character key.

backspaceKey(view) - Called when the user presses the backspace key.

tabKey(view) - Called when the user presses the tab key.

backTabKey(view) - Called when the user presses the back tab key.

shiftArrowRight(view) - Called when the user presses the right arrow key while holding the shift key.

shiftArrowLeft(view) - Called when the user presses the left arrow key while holding the shift key.

shiftArrowUp(view) - Called when the user presses the up arrow key while holding the shift key.

shiftArrowDown(view) - Called when the user presses the down arrow key while holding the shift key.

homeKey(view) - Called when the user presses the home key.

endKey(view) - Called when the user presses the end key.

deleteKey(view) - Called when the user presses the delete key.

enterKey(view) - Called when the user presses the enter key.

selectKey(view) – This event is for the selectKey which is only available on the nSpire calculator. If the view provides a handler for selectKey(), then selectKey(view) will be called. If the view does not provide a handler for selectKey(), but the view does provide a handler for enterKey(), then enterKey(view) will be called instead of selectKey().

escapeKey(view) - Called when the user presses the escape key.

cut(view) - Called when the user chooses the Cut operation.

copy(view) - Called when the user chooses the Copy operation.

paste(view) - Called when the user chooses the Paste operation.

The example below shows handling for handleUserAction(), and for the enterKey() event.

```
1.  function ModuleView:handleUserAction(view)
2.      if view.handleUserAction then
3.          view:handleUserAction()
4.      end
5.  end
6.
7.  function ModuleView:enterKey(view)
8.      if view.enterKey then
9.          view:enterKey()
10.     end
```

```
11. end
12. 
13. function MainPageView:handleUserAction()
14.     --Take special actions for this view.
15. end
16. 
17. function MainPageView:enterKey()
18.     --Take special action for the enter key event.
19. end
```

22. Widget Manager

The WidgetMgr class provides a standardized method for creating and instantiating user interface widgets and relays timer events to the widget timer handler. Each widget type must first be registered with the Widget Manager. Then, each new instance of a widget is created. Finally, each widget must be added to the list of available widgets.

app.frame.widgetMgr:register(widgetName, functionName) – The programmer calls this function to register a widget type. Built-in widgets such as Button have already been registered by the framework. As an example, if the programmer creates a widget class called MyWidget, the programmer will call app.frame.widgetMgr:register("MYWIDGET", function(name, options) local widget = MyWidget(name); self.widgetMgr:addWidget(widget, options) return widget end). This action will register the widget type using the name "MYWIDGET" as an index, and provides the function that will be used when later calling newWidget() to create an instance of the widget.

app.frame.widgetMgr:newWidget(widgetType, widget, ...) – The programmer calls this function to create an instance of the class. For example, calling app.frame.widgetMgr:newWidget("MYWIDGET", "mywidget1", {initFontSize = 9, fontStyle = "r", fontColor = app.graphicsUtilities.Color.blue }) will invoke the function previously registered for widget type "MYWIDGET", add the widget to the WidgetMgr list, and return a handle to the widget. The parameter "mywidget1" is a unique index name for the WidgetMgr list of active widgets.

app.frame.widgetMgr:addWidget(widget, options) – The programmer calls this function to add the widget to the list of WidgetMgr active widgets. For example, app.frame.widgetMgr:addWidget("mywidget1", {initFontSize = 9, fontStyle = "r", fontColor = app.graphicsUtilities.Color.blue }) will add mywidget1 to the list of active widgets and will set the initFontSize, fontStyle, and fontColor properties of that widget. These property values may always be changed later by accessing the property directly. Providing a list of properties in the call to addWidget() is optional and is provided simply as a convenience for the programmer.

app.frame.widgetMgr:newWidget(widgetType, widget, …) - The parameters are:

widgetType – This is an index into the list of registered widgets. Framework built-in widget types are: "BUTTON", "FOOTER_PANE", "SCROLL_PANE", "SIMPLE_STRING", "FEEDBACK", "TEXTBOX", "CIRCLE", "RECTANGLE", "SYMBOL", "LINE", "IMAGE", "TRIANGLE", "FRACTION", "MIXED_NUMBER_INPUT", "DROPDOWN_BOX", "PARAGRAPH", "DIALOG_BOX", "TOOLTIP", "SIMPLE_GRID", and "DEPICTION".

widget – This is a unique string name for this widget for this page.

... - The ellipses receive an array of additional parameters that are specific to the widgetType.

return – The return from the function is the new widget object.

The example below shows how to register a widget class called MyWidget and create a new widget.

```
1.  function ModuleController:begin()
2.      app.frame.widgetMgr:register("MYWIDGET", function(name,
    options) local widget = MyWidget(name);
    app.frame.widgetMgr:addWidget(widget, options) return widget
    end)
3.  end
4.
5.  function MainPageView:stateInit()
6.      self.myWidget1 =
    app.frame.widgetMgr:newWidget("MYWIDGET","myWidget1")
7.  end
```

23. Custom Widgets

The Widget class provides basic and common functionality for all widgets. To create a custom widget, the programmer will typically derive a custom widget from the Widget class and override certain functions, and provide custom functions, as shown in the example below.

```
1.  MyWidget = class(Widget)
2.
3.  function MyWidget:init(name)
4.      -- initialization code here.
5.  end
6.
7.  function MyWidget:paint(gc)
8.      -- overrides the default paint() function provided by the
    Widget class.
9.  end
10.
11. function MyWidget:customFunction()
12.    -- Performs functionality customized for this widget.
13. end
```

To register the custom widget, the programmer should call app.frame.widgetMgr:register() during the Framework call to ModuleView:registerWidgets(), as shown below.

```
1.  function ModuleView:registerWidgets()
2.     app.frame.widgetMgr:register("MYWIDGET", function(name,
    options) local widget = MyWidget(name);
    app.frame.widgetMgr:addWidget(widget, options) return widget
    end)
```

```
3.  end
```

24. Timer

The Timer class provides a wrapper for the TI-nspire timer table. The clock tick speed is set by the Framework to be one tick every 1/10 of a second. The built-in Framework timers are: "POSTMESSAGETIMER", "FRAMETIMER", "MENUTIMER", and "MENUARROWTIMER" and should not be started or stopped by the programmer. The programmer may register additional timer IDs with the Framework by calling app.model:registerTimerIDs(names), where 'names' is an array of quoted strings.

Below is a list of Timer functions available for the programmer.

start(timerID) – Call this function with a timerID to start the timer.

stop(timerID) – Call this function with a timerID to stop the timer.

getMilliSecCounter() – Call this function to get the system timer count in milliseconds.

As an example, the programmer may wish to create a timer called "SLEEPTIMER" that tracks user activity and displays a screen message when the user has been inactive for a while. The programmer first registers the timer ID and a function handler in ModuleModel:configure(). Then, during the initialization state, the programmer may call app.timer:getMilliSecCounter() to get and store the current system timer count and then start the timer. Next, inside the function handler for "SLEEPTIMER", the programmer can check the time difference and act as necessary.

```
1.  function ModuleModel:configure()
2.      app.model:registerTimerIDs({"SLEEPTIMER"}
3.      app.model:registerTimerFunctions({
           [app.model.timerIDs.SLEEPTIMER]
           = function()
           app.controller.pages[app.model.pageIDs.SLEEP_PAGE]:
           handleSleepTimer() end})
4.  end
5.
6.  function SleepPage:stateInit()
7.      self.startTime = app.timer:getMilliSecCounter()
8.      app.timer:start(app.model.timerIDs.SLEEPTIMER)
9.  end
10.
11. function SleepPage:handleSleepTimer()
12.     if app.timer:getMilliSecCounter() - self.startTime >= 5000
           then
13.         -- Take action after 5 seconds.
14.         app.timer:stop(app.model.timerIDs.SLEEPTIMER)
15.     end
16. end
```

25. ModuleTransition

The framework contains a Transition class that can display a transition message when the framework determines that the platform is slow, or when the programmer determines that a transition is needed and sets the app into the TRANSITION state. The ModuleTransition class is an optional class provided by the programmer. If supplied by the programmer, the app.frame.transition.moduleTransition instance is created when the framework creates the frame.

ModuleTransition is a framework view class that is used to display information during a transition from one state to another state, as needed. Not all state changes need to use the ModuleTransition class. An example of when a transition is needed is when the user completes a problem, and then the next problem is being loaded. The transition might display, "Problem 2" for a few seconds. The transition painting is done on the current view, not the upcoming view.

During the creation of the frame, the framework will call app.framework.transition:setupTransition(). The framework will then call app.framework.transition.moduleTransition:setupTransition() if this method is provided by the programmer. The programmer may then call app.frame.transition:addObject(widget) for each display widget that should be painted during the transition.

During a transition, the framework will call app.framework.transition:modifyView(pageID), where pageID is the page being transitioned to. The framework will then call app.framework.transition:moduleTransition:modifyView(pageID) if this method is supplied by the programmer. This function allows the programmer to modify the content of the displayed transition messages.

The example below shows how to set up a transition, how to modify the text that is displayed, and how to cause the dispatcher to enter into the TRANSITION state from the END state.

```
1.  function ModuleTransition:setupTransition()
2.     local transitionMsg1 =
    app.frame.widgetMgr:newWidget("SIMPLE_STRING", "transitionMsg1",
    {initFontSize = 16, fontStyle = "r", fontColor =
    app.graphicsUtilities.Color.blue, visible = false,
                          initSizeAndPosition = {0, 0, .5, .5} } )
3.     app.frame.transition:addObject(transitionMsg1)
4.  end
5.
6.  function ModuleTransition:modifyView(nextPageID)
7.      local transitionMsg1 =
    app.frame.transition.objects["transitionMsg1"]
8.     transitionMsg1:modifyProperties(text = "Transition", visible
    = true} )
9.  end
10.
11. function ModuleController:stateEnd(page)
12.    page.nextState = app.model.statesList.TRANSITION
13.    app.controller:dispatch(app.controller.activePageID)
14. end
```

26. Utilities

The Utilities class provides miscellaneous functions.

function Utilities:queueAdd(tbl, d) – Call this function with some data (d) to add to table (tbl) at the end in queue fashion.

function Utilities:queueRemove(tbl) – Call this function to retrieve the first item in a queue from a table (tbl) and have the data returned.

function Utilities:push(tbl, d) – Call this function with some data (d) to add to table (tbl) at the top in stack fashion.

function Utilities:pop(tbl) – Call this function to retrieve the top item in a stack from a table (tbl) and have the data returned.

27. GraphicsUtilities

The GraphicsUtilities class provides functions for working with graphics.

GraphicsUtilities.Color – This table contains names for commonly used colors. Available colors are: black, white, red, green, blue, blue2, red2, pink, yellow, orange, purple, lightpurple, silvergrey, whitegrey, darkgrey, grey, verydarkgrey, darkblue, pastelyellow, pastelgreen, pastelblue, tooltipgreen, yellowish, orangeish, redish, purpleish, cyanish, cyanish2, turquoise, thistle, maizeyellow, brightyellow, harvestgold, trueblue, babyblue.

GraphicsUtilities.figureTypeIDs – This table contains IDs for drawing types. Available figure types are: RIGHT_ARROWHEAD, LEFT_ARROWHEAD, DOWN_ARROWHEAD, UP_ARROWHEAD, CHECK_MARK, TICK_MARK, SMALL_RIGHT_ARROW, SMALL_LEFT_ARROW, SMALL_DOWN_ARROW, SMALL_UP_ARROW, CIRCLE, TRIANGLE, STAR, DIAMOND, SQUARE, UP_ARROW, DOWN_ARROW, LEFT_ARROW, RIGHT_ARROW, HOUR_GLASS.

GraphicsUtilities.drawStyles – This table contains the drawing styles for widgets such as the DropDown Box. Available drawStyles are: FILL_AND_OUTLINE, FILL_ONLY, OUTLINE_ONLY.

GraphicsUtilities.figureObjects – This table contains the functions for creating new figure objects. Built-in objects are: CIRCLE, STAR, TRIANGLE, SQUARE, DIAMOND, UP_ARROW, DOWN_ARROW, LEFT_ARROW, RIGHT_ARROW, HOUR_GLASS. The programmer may add custom objects to this table.

function GraphicsUtilities:drawFigure(figureType, gc, x, y, scale, color, style) – Call this function to draw an available figure type. Pass in the figureTypeID, the graphics context, the x,y position, a scale factor, the color and a drawing style.

function GraphicsUtilities:newFigure(figureType, name, …) – Call this function to create a new instance of a figure type. Pass in the figureTypeID, a unique name for this instance, and any additional parameters required for creating a particular figure.

The code snippet below shows examples of how to use the GraphicsUtilities class.

```
1.  function ModuleModel:configure()
2.      app.graphicsUtilities.figureTypeIDs =
           app.enum({"MY_FIGURE"},
           app.graphicsUtilities.figureTypeIDs)
3.      app.graphicsUtilities.figureObjects
           [app.graphicsUtilities.figureTypeIDs.MY_FIGURE] =
           function(name, options) local widget = MyFigure(name);
         if options ~= nil then widget:modifyProperties(options)
         end return widget end
4.  end
5.
6.  function MyPageView:paint(gc)
7.       app.graphicsUtilities:drawFigure(app.graphicsUtilities.
             figureTypeIDs.MY_FIGURE, gc, x+.5*w-7*scale,
             y+.5*h-6*scale, 2*scale,
             app.graphicsUtilities.Color.black,
             app.graphicsUtilities.drawStyles.FILL_ONLY)
8.  end
```

28. StringTools

The StringTools class provides functions for working with strings.

StringTools:scaleFont(fontSize, scaleFactor) – Call this function to retrieve a new font size based on a scaleFactor. Pass in the fontSize and the screen scaleFactor and the return will be a new fontSize that is appropriate for the given screen scale.

StringTools:trim(s) – Call this function to remove leading and trailing whitespace from a string. Pass in the string and the trimmed string is returned.

StringTools:removeWhiteSpace(s) – Call this function to remove leading, internal, and trailing whitespace from a string. Pass in the string and a string is returned with no whitespace.

StringTools:splitInTwo(s, m) – Call this function with a string, s, and a character, m. The string will be split into two separate strings at character m, and the left side and right side will be returned as two separate strings.

StringTools:splitInThree(s, leftMarker, rightMarker) – Call this function with a string, s, a character, leftMarker, and a second character, rightMarker. The string will be split into three separate strings at characters leftMarker and rightMarker and three separate strings will be returned.

StringTools:uFind(s, c, startIdx, plain, regularFind) – Call this function to locate a Unicode character. Pass in a string, s, the Unicode character, c, and the starting position in the string, startIdx. Both plain = true and regularFind = true if this function should use string.find().

StringTools:reverseFind(s, pattern, index, plain) – Call this function to locate a string pattern, starting from the right side of the string, s. index is the position of string, s, to start the search. plain = true for regular searching.

StringTools:getNumOfChars(str) – Call this function to return the number of characters in a string, str.

StringTools:getStringWidth(s, fontFamily, fontStyle, fontSize) – Returns the string width in pixels without requiring a gc.

StringTools:getCharacterWidth(c, fontFamily, fontStyle, fontSize) – Returns the character width in pixels without requiring a gc.

StringTools:getCharacterHeight(c, fontFamily, fontStyle, fontSize) – Returns the character height in pixels without requiring a gc.

StringTools:getStringWidthHeight(s, fontFamily, fontStyle, fontSize, gc) – Calculates and returns the string width and height in pixels.

StringTools:getCharacterWidthHeight(c, fontFamily, fontStyle, fontSize, gc) – Calculates and returns the character width and height in pixels.

StringTools:centerString(width, s, fontFamily, fontStyle, fontSize) – Call this function to center a string within the provided width.

29. Rationals

The Rationals class provides functions for working with rational numbers.

Rationals:simplifyDecimal(s) – Call this function to reduce a decimal string, s, to its simplest format by removing leading and trailing zeros, and removing a leading plus sign and trailing decimal point. The simplified decimal string is returned.

Rationals:simplifyRational(s) – Call this function to reduce any rational number (integer, fraction, or decimal) into its simplest form by reducing fractions, removing leading and trailing zeros, leading plus sign and trailing decimal point. The simplified rational string is returned.

Rationals:findDecimalPlaceValue(s) – Call this function to return the place value of the digit farthest to the right of the decimal point. Returns 0 for whole number, 1 for tenths place, etc.

Rationals:GCD(a, b) – Call this function to return the Greatest Common Divisor of two whole numbers.

Rationals:LCM(a, b) – Call this function to return the Lowest Common Multiple of two whole numbers.

Rationals:toFraction(str, allowReducing) – Call this function to convert a decimal number into a fraction. Pass in the decimal value as a string, str. allowReducing is true if the returned fraction should be simplified. The return value is a fraction in string format. A fraction with a 1 as the denominator is returned as a whole number.

Rationals:toImproper(s) – Call this function to convert a mixed number to improper format. Pass in a mixed number such as “5 1/2" and the return will be “11/2”.

Rationals:toDecimal(s) – Call this function to convert a fraction to a decimal. For example, pass in “3/4” and the function will return 0.75 as a string.

Rationals:simplifyMixed(s, allowReducing, simplestFormat, makeProper, makeMixed) – Call this function to convert a mixed number to improper format. allowReducing = true means to reduce fractions. simplestForm = true means to simplify “0 ½” to “1/2”. makeProper = true means “5/2” = “2 ½”. makeMixed = true means “1/2” = ”0 ½”.

Rationals:reduceFraction(s) – Call this function to reduce a fraction in string form to its simplest form. “4/12” will be returned as “1/3”.

Rationals:isInteger(x) – Returns true if the passed in number value, x, is an integer.

Rationals:isFraction(s) – Returns true if the passed in string value, s, is formatted as a valid fraction such as “2/3”. The first return parameter is true if the passed in value is a fraction. The second return parameter is the passed in string with whitespace removed.

Rationals:cleanupMath(x) – A Lua math operation such as 1 divided by 2 may return a value such as 0.5000000001 instead of just 0.5. Call this function after any Lua math function call to ensure the last extra bits are removed. The trimmed return number value is to the 1/10,000 place.

Rationals:getFactors(n) – Returns a table of values consisting of the whole number factors of the passed in whole number value n.

Rationals:getPrimeNumbers(startNumber, endNumber) – Returns a table of prime numbers between and including startNumber and endNumber.

Rationals:getCompositeNumbers(startNumber, endNumber) – Returns a table of composite numbers between and including startNumber and endNumber.

Rationals:add(a, b) – Call this function to add two numbers and return the result after a call to Rationals:cleanupMath().

Rationals:subtract(a, b) – Call this function to subtract b from a and return the result after a call to Rationals:cleanupMath().

Rationals:multiply(a, b) – Call this function to multiply two numbers and return the result after a call to Rationals:cleanupMath().

Rationals:divide(a, b) – Call this function to divide two numbers and return the result after a call to Rationals:cleanupMath().

Rationals:raise(a, b) – Call this function to raise a to the power of b and return the result after a call to Rationals:cleanupMath().

Rationals:mod(a, b) – Call this function to find the modulo a mod b and return the result after a call to Rationals:cleanupMath().

Rationals:roundHalfUp(n, decimals) – Call this function to round up a decimal number value, n, to place value, decimals. The rounded up number value is returned.

Rationals:computeChecksum(n) – Call this function to calculate the checksum of all the digits of a number value, n.

Rationals:pointFromDegrees(x0, y0, radius, degrees) – Call this function to calculate the cartesian degrees of a point on a circle. x0, y0 are center of circle in pixels. 'radius' is the length of the radius. 'degrees' is the angle. 0 degrees is on the right side. 90 degrees is at the top. Returns cartesian degrees as two values x and y.