

Civilization 5 UI Tutorial

Part 8, Miscellanea

Version: 1.1

Status: Draft

Date: 23 May 2012

Author: William Howard

Contents

Overview	3
Context Show/Hide Handling.....	3
City View Handling	7
Keyboard Event Handling.....	7
Mouse Event Handling	8
Text Truncation & Wrapping.....	9
Focus Control	9
Colours	3
Line Element	9
Timers	12
LuaContext Element.....	12
Summary	14

Overview

The first seven tutorials in this series on the Civ 5 User Interface (UI) controls have focused on the XML elements used to construct the various interface components, with just enough Lua to get them to work. This tutorial wraps up with some of the less frequently used, but nonetheless useful, XML elements and attributes and also covers additional Lua code that occurs within most contexts to provide additional functionality.

Unlike the other tutorials, not every section of this tutorial has specific code, as the discussed feature can be added to examples in the previous tutorials.

All the code in this tutorial can be downloaded from the in-game ModHub as "Test - UI Tutorial - 8 Miscellanea" found under the "Other" category.

So, to start we need a ModBuddy project ...

Create a Misc Mod

Using ModBuddy, create a new mod called "Test - Misc" (or some such).

To this mod add the two folders "UI" and "XML". In the UI folder create the two files "Misc.xml" and "Misc.lua" (delete the standard content added to these files). In the XML folder create the file "MiscText.xml" (you can leave the standard content in this file).

In the mod's properties, on the "Mod Info" tab, uncheck "Affects Saved Games". On the "Actions" tab, add an "On Mod Activated - Update Database" entry for "XML/MiscText.xml". On the "Content" tab, add an "InGameUIAddin" for "UI/Misc.xml".

Save the project.

Colours

A number of UI elements have a `Color=""` attribute (or relative of) and sometime I've used `Color="255,255,200,255"` and sometimes `Color="Culture,255"` - so what are these numbers and what names can we use?

The format of the `Color=""` attributes are `Color="iRed, iGreen, iBlue, iAlpha"` - where *iRed*, *iGreen*, *iBlue* and *iAlpha* are numbers in the range 0 to 255 and represent the value to assign to that channel.

The core game provides a number of pre-defined colours - Beige, Culture, Red, etc - so we can use `Color="Beige"` as a short-hand for `Color="255,255,200,255"`. We can also override the alpha channel, so if we want a translucent beige we can use `Color="Beige,128"` as a shorthand for `Color="255,255,200,128"`

Confusingly the Lua methods take channel values between 0 and 1, with all four channels in a single set. But the set items are not "r, g, b and a" but "x, y, z and w" - so in Lua "Beige,128" is "{x=1, y=1, z=0.78, w=0.5}"

There is an easier to read (and hence less confusing) way to do this by using the Color() function in FLuaVector

```
include("FLuaVector")
beige128 = Color(255/255, 255/255, 200/255, 128/255)
Controls.EditBorder:SetColor(beige128)
Controls.EditLabel:SetColor(beige128, 0)
```

The pre-defined colours can be found in ColorAtlas.lua, and the more commonly used ones are

- Beige Color="255,255,200,255"
- Black Color="0,0,0,255"
- White Color="255,255,255,255"
- Red Color="255,0,0,255"
- Green Color="0,200,0,255"
- Blue Color="0,0,255,255"
- Science Color="33,190,247,255"
- Gold Color="231,213,0,255"
- Production Color="198,136,50,255"
- Food Color="166,230,75,255"
- Culture Color="231,0,231,255"

(The pre-defined colour sets are also in that file, but the UI typically only uses "Beige_Black_Alpha".)

Color0, Color1, ColorSets and Color2

"Box" elements (Box, Grid, Image, etc) can have a single background colour, so take the Color attribute to define this. "Text" elements (Label, TextButton, EditBox, etc) need both a foreground and a background colour, so use the ColorN attributes.

Add the following to the "UI/Misc.xml" file

```
<?xml version="1.0" encoding="utf-8" ?>
<Context ID="Misc1">
  <Box Size="400,270" Anchor="C,C" Color="White,255" ConsumeMouse="1">
    <Box Size="398,268" Anchor="C,C" Color="Black,255"/>

    <Stack Anchor="C,C" StackGrowth="Bottom" Padding="8">
      <Label Anchor="C,T" Font="TwCenMT24" FontStyle="Shadow"
ColorSet="Beige_Black_Alpha" String="TXT_KEY_TEST_MISC_MESSAGE"/>

      <Label Anchor="C,T" Font="TwCenMT24" FontStyle="Shadow"
Color0="Beige" Color1="Black_Alpha"
String="TXT_KEY_TEST_MISC_MESSAGE"/>

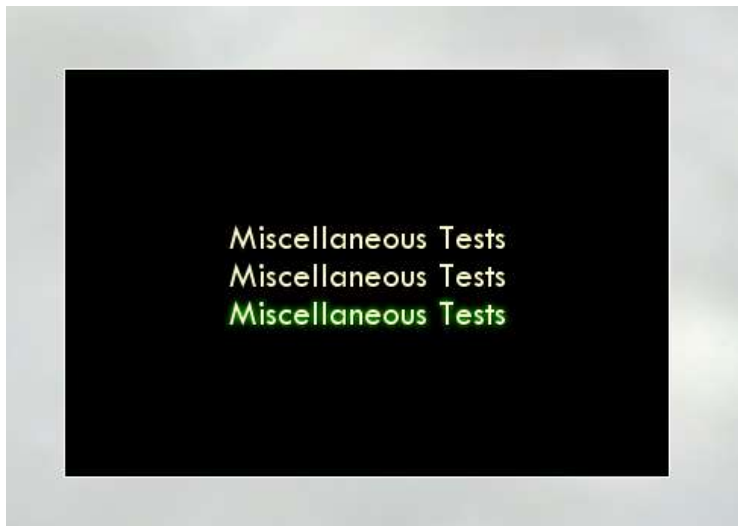
      <Label Anchor="C,T" Font="TwCenMT24" FontStyle="SoftShadow"
ColorSet="Beige_Black_Alpha" Color2="Green"
String="TXT_KEY_TEST_MISC_MESSAGE"/>
    </Stack>
  </Box>
</Context>
```

```
</Stack>
</Box>
</Context>
```

Add the following to the "XML/MiscText.xml" file

```
<?xml version="1.0" encoding="utf-8"?>
<GameData>
  <Language_en_US>
    <Row Tag="TXT_KEY_TEST_MISC_MESSAGE">
      <Text>Miscellaneous Tests</Text>
    </Row>
  </Language_en_US>
</GameData>
```

Save the files and build the mod. Start Civ 5, enable the mod, and start a new game.



```
<Label Anchor="C,T" Font="TwCenMT24" FontStyle="Shadow"
ColorSet="Beige_Black_Alpha" String="TXT_KEY_TEST_MISC_MESSAGE"/>

<Label Anchor="C,T" Font="TwCenMT24" FontStyle="Shadow" Color0="Beige"
Color1="Black_Alpha" String="TXT_KEY_TEST_MISC_MESSAGE"/>

<Label Anchor="C,T" Font="TwCenMT24" FontStyle="SoftShadow"
ColorSet="Beige_Black_Alpha" Color2="Green"
String="TXT_KEY_TEST_MISC_MESSAGE"/>
```

The first line uses a ColorSet attribute, which is really just shorthand for Color0="Beige" Color1="Black_Alpha" as can be seen in the second line.

Because the FontStyle of the third line is SoftShadow, we need to specify the "smudge" colour, which we do via the Color2 attribute (green in this case so you can see it, but usually Beige).

There are also ColorLayer0 and ColorLayer1 attributes. These seem to be synonyms for Color0 and Color1. However, RadioButtons and CheckBoxes only use ColorLayer*N* attributes so you may need to use these attributes with these elements.

Full Screen Size

Sometimes we want to create an element which is the full size of the screen (width, height or both). We could create the element with an arbitrary size and then use Lua to resize it

```
<!-- We will resize this box with Lua to the full screen area -->
<Box ID="FullScreenBox" Size="100,100" Color="Black,200"/>

local iScreenX, iScreenY = UIManager:GetScreenSizeVal()
Controls.FullScreenBox:SetSizeVal(iScreenX, iScreenY)
```

but there is an easier (and more readable) way

```
<Box ID="FullScreenBox" Size="Full,Full"/>
```

You can also offset the area, we could re-write BGBlock_ClearTopBar as

```
<Box Size="Full,30" Anchor="C,T" Color="0,0,0,0" ConsumeMouse="0">
  <Box Size="Full,Full" Offset="0,30" Anchor="C,T" Color="0,0,0,200"
  ConsumeMouse="1"/>
</Box>
```

Context Show/Hide Handling

Sometimes we want to know when our dialog is being shown (or hidden) as we may need to initialise controls (see Focus Control below), we can detect this by hooking the ShowHideHandler

```
function ShowHideHandler(bIsHide, bIsInit)
  if (bIsInit) then
    -- Perform one time only initialisation
  end

  if (bIsHide) then
    -- Being hidden, destroy anything we don't need
  else
    -- Being shown, initialise prior to the user seeing the context
  end
end

ContextPtr:SetShowHideHandler(ShowHideHandler)
```

City View Handling

As mentioned in passing in Tutorial 6, if we have any "informational" UI elements (for example, those developed in Tutorial 5) they probably shouldn't show when the user is in the City View. By hooking the City View Enter/Exit events we can easily hide/show our context (and hence the informational elements)

```
function OnEnterCityScreen()
    ContextPtr:SetHide(true)
end
Events.SerialEventEnterCityScreen.Add(OnEnterCityScreen)

function OnExitCityScreen()
    ContextPtr:SetHide(false)
end
Events.SerialEventExitCityScreen.Add(OnExitCityScreen)
```

Keyboard Event Handling

All of our dialogs have relied on the user clicking the OK button to close them. Good practice dictates that our dialogs should also respond to the keyboard - specifically the Return/Enter keys to signify "Yes" or "OK" and/or the Escape key to signify "No" or "Close". For a dialog with a single "OK"/"Close" button both Return and Escape should close the dialog.

We can achieve this by registering an input handler that performs the same actions as the UI button(s)

```
function InputHandler(uiMsg, wParam, lParam)
    if (uiMsg == KeyEvents.KeyDown) then
        if (wParam == Keys.VK_ESCAPE) then
            OnClose() -- or OnNo()
        elseif (wParam == Keys.VK_RETURN) then
            OnOK() -- or OnYes()
        end
    end

    return true
end
ContextPtr:SetInputHandler(InputHandler)
```

the only thing we need to note about the InputHandler is that if we process the key press, the function must return true (otherwise the key press will be passed to any controls under our dialog, which at the least will be the World Map or City View). If we don't process the key, we can return false (or just drop-off the end of the function, which seems to be the de-facto standard).

In addition to the Return/Enter and Escape keys, we can also detect other keys via the following Keys

- **Keys.A thru Keys.Z** - the letter keys, if you need to differentiate between upper and lower case, or with the Control or Alt key held down, perform the test in combination with the `UIManager:GetShift()`, `:GetAlt()` and `:GetControl()` functions
- **Keys.VK_LEFT, Keys.VK_RIGHT, Keys.VK_UP, Keys.VK_DOWN** - the cursor keys
- **Keys.VK_NEXT, Keys.VK_PRIOR** - the PgUp and PgDn keys
- **Keys.VK_OEM_MINUS, Keys.VK_OEM_PLUS** - the - and + keys on the number pad
- **Keys.VK_NUMPAD0 thru Keys.VK_NUMPAD9** - the 0 to 9 keys on the number pad
- **Keys.VK_OEM_3** - the ~ key
- **Keys.VK_SPACE** - the space bar
- **Keys.VK_F1 thru Keys.VK_F10** - the F1 to F10 keys (VK_F11 and VK_F12 may work, but not all keyboards have them)

Mouse Event Handling

The input handler also receives mouse events

```
function InputHandler(uiMsg, wParam, lParam)
    if (uiMsg == MouseEvents.MouseLButtonUp) then
        -- Do something on the left mouse button being released
        return true
    elseif (uiMsg == MouseEvents.MouseRButtonDown) then
        -- Do something on the right mouse button being pressed
        return true
    end
end
ContextPtr:SetInputHandler(ProcessInput)
```

again, if you use the event return true, otherwise just drop-off the end of the function.

We can also detect mouse movement

```
function InputHandler(uiMsg, wParam, lParam)
    if (uiMsg == MouseEvents.MouseMove) then
        local dx, dy = UIManager:GetMouseDelta()
        if ((dx + dy) ~= 0) then
            -- The mouse moved, typically hide something
        end
    end
end
ContextPtr:SetInputHandler(ProcessInput)
```

if you don't want anything else to know about the mouse move return true, but typically we just drop-off the end of the function to let the World View or City View handle the mouse move as well.

We can also get the mouse position at any time via

```
local iMouseX, iMouseY = UIManager:GetMousePos()
```

However, we rarely want to know where the mouse is on the screen, more often where the mouse is on the map, and we can get that via

```
local pPlot = Map.GetPlot(UI.GetMouseOverHex())
```

Text Truncation & Wrapping

If we are adding text of unknown length to a label (eg by constructing a string from the current player data, or from player entered text) it may exceed the available screen space. By using either the `TruncateWidth=""` attribute on the Label element, or by calling the `:SetTruncateWidth()` method on the label control, we can ensure that the text will not overflow the end of the label.

```
<Label Size="20,100" TruncateWidth="100" ...>  
:SetTruncateWidth(100)
```

If the text string is too long to fit, the core game engine will truncate it and append three dots (ellipses) to the end of the text, eg, "This text is way too long to fit" might become "This text is way too lon ..."

If the Label is tall enough to take more than one line, we can force line breaks into the text string with "[NEWLINE]". However, sometimes we don't know where the line breaks need to be in advance (for example, the descriptions of the civilization on the game loading screen) and we'd just like the text to "flow" into the label. We can achieve this by using the `WrapWidth=""` attribute on the Label element (there doesn't seem to be a corresponding Lua method)

```
<Label Size="100,500" WrapWidth="500" ...>
```

Focus Control

In Tutorial 4 we looked at `EditBox` elements and covered the `FocusStop` attribute. Using Lua we can also force a specific `EditBox` to have the input focus when the dialog is displayed, for example,

```
function ShowHideHandler(bIsHide, bIsInit)  
    if (not bIsHide) then  
        Controls.NotifyHeading:TakeFocus()  
    end  
end  
ContextPtr:SetShowHideHandler(ShowHideHandler)
```

AnchorSide

In Tutorial 1 Dialog 4, when we added decoration to the dialog box, we briefly mentioned the `AnchorSide` attribute

"The `AnchorSide` attribute determines if the image is placed on the "inside" or "outside" of the grid (and further confuses the notion of "up", "down", "left" and "right"). For most decoration all we really need to do is find a standard dialog that has the decoration we want and copy it - understanding is not a requirement!"

Change the the "UI/Misc.xml" file to

```
<?xml version="1.0" encoding="utf-8" ?>
<Context ID="Misc2">
  <Box Size="400,270" Anchor="C,C" Color="White,255" ConsumeMouse="1">
    <Box Size="398,268" Anchor="C,C" Color="Black,255"/>

    <Label ID="CC" Anchor="C,C" Font="TwCenMT24" FontStyle="Shadow"
ColorSet="Beige_Black_Alpha" String="TXT_KEY_TEST_MISC_MESSAGE"/>

    <Label ID="CT" Anchor="C,T" Font="TwCenMT20" FontStyle="Shadow"
ColorSet="Beige_Black_Alpha" String="C,T"/>
    <Label ID="RT" Anchor="R,T" Font="TwCenMT20" FontStyle="Shadow"
ColorSet="Beige_Black_Alpha" String="R,T"/>
    <Label ID="RC" Anchor="R,C" Font="TwCenMT20" FontStyle="Shadow"
ColorSet="Beige_Black_Alpha" String="R,C"/>

    <Label ID="CTO" Anchor="C,T" AnchorSide="O,O" Font="TwCenMT20"
FontStyle="Shadow" ColorSet="Red_Black" String="C,T O,O"/>
    <Label ID="RTOO" Anchor="R,T" AnchorSide="O,O" Font="TwCenMT20"
FontStyle="Shadow" ColorSet="Red_Black" String="R,T O,O"/>
    <Label ID="RTOI" Anchor="R,T" AnchorSide="O,I" Font="TwCenMT20"
FontStyle="Shadow" ColorSet="Green_Black" String="R,T O,I"/>
    <Label ID="RTIO" Anchor="R,T" AnchorSide="I,O" Font="TwCenMT20"
FontStyle="Shadow" ColorSet="Gold_Medal" String="R,T I,O"/>
    <Label ID="RCO" Anchor="R,C" AnchorSide="O,O" Font="TwCenMT20"
FontStyle="Shadow" ColorSet="Red_Black" String="R,C O,O"/>
  </Box>
</Context>
```

(The downloadable code has labels at all 9 Anchor points, but I've omitted the "left" and "bottom" ones for brevity.)

Add the following to the "UI/Misc.lua" file

```
Controls.RT:SetOffsetVal(7,7)
Controls.RTOO:SetOffsetVal(7,7)
Controls.RTIO:SetOffsetVal(7,7)
Controls.RTOI:SetOffsetVal(7,7)
```

Save the files and build the mod. Start Civ 5, enable the mod, and start a new game.



The default AnchorSide is "I,I" for "inside X and inside Y", which can be seen from the placement of the white text Labels being "inside" the Box. All the red text has been placed "outside" (AnchorSide="O,O") the box, and appears as you would expect.

The yellow and green text (top right corner) shows that you can also place text "inside X, outside Y" and "outside X, inside Y" respectively.

The four Labels in the top right corner have all been offset (via Lua) by +7 in each direction - as you can see this has moved them "away" from the corner. So when we said

"Like most things in life, the answer is "it depends". In this case it depends on where the anchor is. Because the label has an anchor of "C,C" "negative y" is up the screen, whereas the gridbutton has an anchor of "C,B" so "negative y" is down the screen."

This should really be

"Like most things in life, the answer is "it depends". In this case it depends on where the anchor is *and on what the AnchorSide attribute is.*"

Line Element

If you followed the code in the Percent Meter sample in Tutorial 5 you will have noted that I omitted to cover a UI element - namely the Line element

```
<Line Color="Red" Start="1,1" End="100,100" Width="1" />
```

which pretty much does what it says! The Anchor and Offset attributes can also be specified. Lines can also be moved with the :SetStartVal(iStartX, iStartY) and :SetEndVal(iEndX, iEndY) methods.

Timers

Occasionally, we need to update some element of our context at intervals, for example a clock or a "processing" meter. While we can't get updates at regular or pre-determined intervals, we can receive notifications at intervals by registering an "OnUpdate" handler

```
function OnUpdate(fDeltaTime)
    -- Do something
end
ContextPtr:SetUpdate(OnUpdate)
```

fDeltaTime is the time (in seconds) since we were last called.

To stop receiving notifications use the :ClearUpdate() method. For example, to delay 10 seconds and then do something use

```
local fTimeLimit

function OnUpdate(fDeltaTime)
    fTimeLimit = fTimeLimit - fDeltaTime

    if (fTimeLimit < 0.0) then
        ContextPtr:ClearUpdate()

        -- Do something
    end
end

fTimeLimit = 10.0
ContextPtr:SetUpdate(OnUpdate)
```

For an interesting example of a variation of this technique see how the image is zoomed in WonderPopup.lua

LuaContext Element

It is possible for one context to include another (and this is how the entire UI is built, the InGame context includes the Civilopedia context and the WorldView context (among many others) and the WorldView includes the MiniMap, InfoCorner, DiploCorner etc contexts)

If we're building a complex dialog, for example a dialog with "tabs" where each tab shows a different (complex) set of controls (very much like the Diplo Overview dialog which has tabs for "Our Relationships", "Current Deals" and "Global Relationships"), rather than having one (massive)

context with all the controls for all tabs, we can split the controls for each tab into their own context and then include the sub-contexts into the main one.

To do this we use the `LuaContext` element

```
<LuaContext FileName="MyDialogTab1" ID="MyDialogTab1Panel" Hidden="1"
Anchor="L,T" Offset="20,80" />
```

We can then use `Controls.MyDialogTab1Panel:SetHide()` to show/hide the entire set of controls depending on which tab the user selects.

Summary

This tutorial has detailed aspects of the Civ 5 User Interface (UI) elements and associated Lua methods not covered in the other tutorials in this series.

All of the XML and Lua code for the examples in this tutorial can be downloaded from the Mod Hub as "Test - UI Tutorial - 8 Miscellanea" (in the Other category). Each miscellaneous step is included separately and can be displayed from the FireTuner Lua Console tab by selecting the Misc context and then entering "ShowN()" in the command line (where N is the step to display, eg "Show1()", "Show2()", etc)