Creating VSTGUI Custom Views 2: *CViewContainer* Custom Views & Caching Custom View Pointers
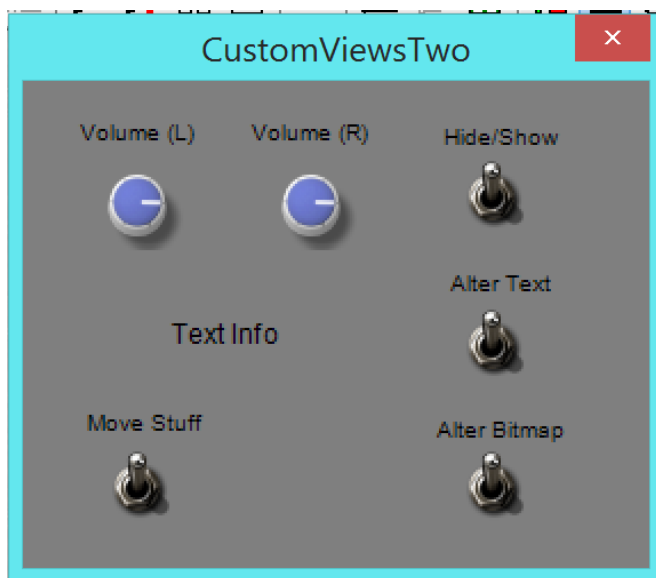
Will Pirkle

In the last module, you learned how to create Custom View objects, either ordinary VSTGUI objects, or your own subclassed versions. In this module, we will continue with two more details: making *CViewContainer* Custom Views and caching view pointers.

In the last module's project *CustomViews*, we created the objects and then returned the pointers. In this module, we will learn how to cache (store) the pointers to use later. With the object pointers, we can control many aspects of the GUI objects from within our code.

> **I am demonstrating this technique due to several requests from RackAFX users for this kind of GUI customization. However, there are many people who will frown upon this kind of use of the GUI object pointers since one of the goals of object oriented design is to keep objects and their implementation details as separate as possible.**

Example 1: Making a *CViewContainer* Custom View and caching the pointer
Open the **CustomViewsTwo** project that accompanies this module, compile the plugin and load it. The GUI for this project contains two volume control knob templates, one text label, and four on-off switches.

We will use the switches to demonstrate how to control the various GUI elements from within the plugin.
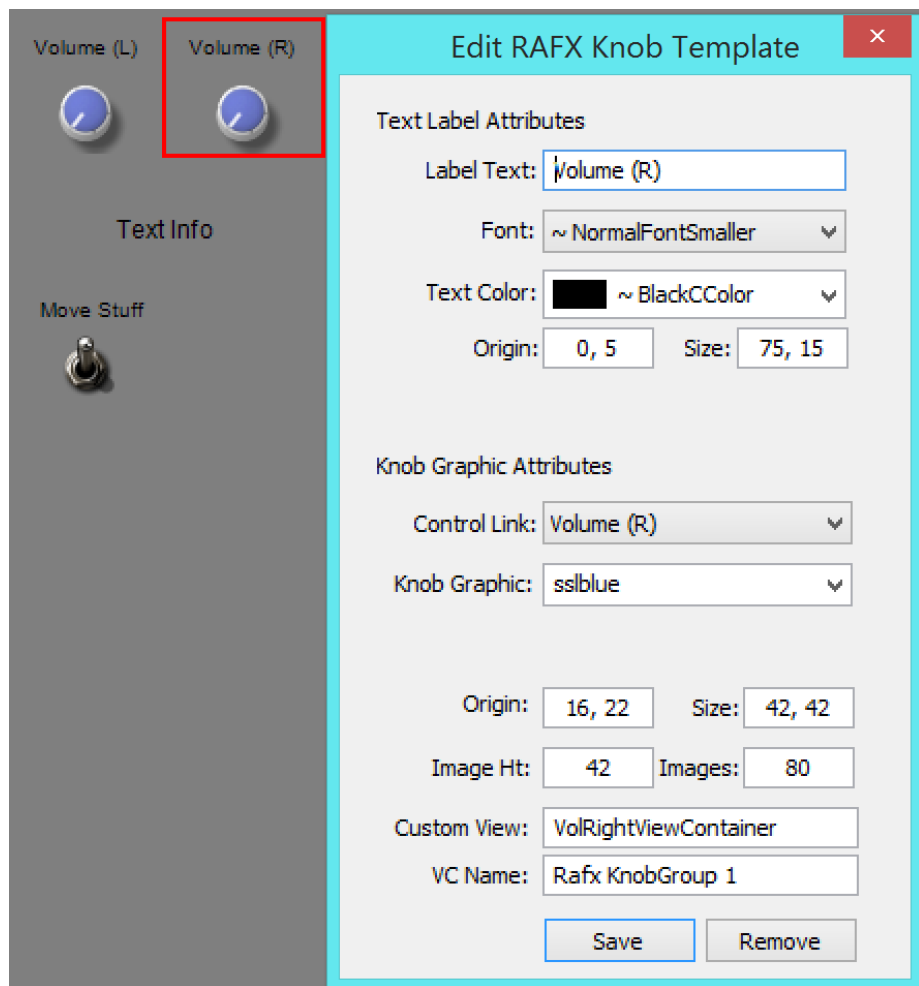
Hide/Show: this will hide or show the Volume Right knob template (both the text label and knob at the same time)

Alter Text: this will change the "Text Info" label in the center of the GUI

Alter Bitmap: this will change the knob bitmap for the left volume control

Move Stuff: this will move the Right Knob Template (both the text label and knob at the same time)

In the GUI Designer, right-click on the right volume knob's template container. At the bottom of the dialog, you can see where to enter a custom view name for the whole view container:

Here, I have named the knob template (i.e. it's *CViewContainer*) custom view as "VolRightView-Container" at the bottom of the dialog.

You can also set custom view names for any *CViewContainer* objects that you add to the GUI Designer, either as part of RackAFX or User Templates, or with the Group Frame drag and drop method.

In the *CustomViewTwo* project, open the project's .h file and you can see several necessary items:

At the top is the #include statement for the vstgui4 library:

```
// --- the entire library is contained in this #include statement
#include "../vstgui4/vstgui/vstgui.h"
```

In the user variable section, we have our GUI helper and also a *CViewContainer** named m_pVolRightVC.

```
    // --- our helper
    CVSTGUIHelper m_GUIHelper;

    // --- cached pointers to custom view objects
    CViewContainer* m_pVolRightVC;
```

When caching pointers is is critical that you follow some rules to make sure you never accidentally try to use a pointer whose object has been deleted by the VTSGUI framework (remember, reference counting allows the framework to delete controls that are not used any more).

Caching Pointer Rules

1. declare your pointer in the .h file of the project
2. set the pointer to NULL in the Constructor
3. add code for the GUI_WILL_CLOSE message in *showGUI()* to NULL out the pointer
4. always check the validity of the pointer before using it

Step through the project's .cpp file to see these rules in action, as well as the instantiation of the view container object:

Constructor:
```
CCustomViewsTwo::CCustomViewsTwo()
{
      // Added by RackAFX - DO NOT REMOVE
      //
      // initUI() for GUI controls: this must be called before initializing/
      // using any GUI variables
      initUI();
      // END initUI()

      <SNIP SNIP SNIP>

      // Finish initializations here

      // --- cached pointers
      m_pVolRightVC = NULL;
}
```

showGUI()

First, the GUI_WILL_CLOSE message:

```
case GUI_WILL_CLOSE:
{
      // --- NULL out pointers - do NOT delete underlying objects
      if(m_pVolRightVC)
            m_pVolRightVC = NULL;

      return NULL;
}
```

We create the object in the GUI_CUSTOM_VIEW message. The pointer is cached after creation in the m_pVolRightVC variable. Notice the call to *setTransparency()* here - we need to do this ourselves as transparency is OFF by default. You can do other view customization here as well.

```
case GUI_CUSTOMVIEW:
{
      // --- decode the customViewName variable
      if(info->customViewName.compare("VolRightViewContainer") == 0)
      {
            // --- get the needed attributes with the helper
            const CRect rect = m_GUIHelper.getRectWithVSTGUIRECT(
```

```
                                        info->customViewRect);

        // --- create the VC
        m_pVolRightVC = new CViewContainer(rect);

        // --- make transparent (could also set background color/bitmap!)
        m_pVolRightVC->setTransparency(true);

        // --- return control cloaked as a void*
        return (void*)m_pVolRightVC;
    }

    etc…
```

Using the cached pointer: hiding and showing GUI elements with *setVisible()*
The switch on the GUI that will show and hide this view container is a RackAFX 2-state switch, which we can map as a *COnOffButton* in VSTGUI. **To use the pointer, we need to trap the message in *check-UpdateGUI()* for this control. This function is called when a GUI control changes.** Ordinarily, you fill our a GUI change structure and pass that back to the host to update the GUI controls. **However, in this case we are modifying the appearance of the control directly.** To do so, we decode the nControlIndex variable, just the same as in *userInterfaceChange( )*, however we use the input variable *fValue* to make our comparisons. The variable *fValue* represents the value that the altered control will take on.

The RackAFX Control ID is 45 so we use the case statement for that and simply call the *setVisible()* method on the view container to show and hide it, and all its contents, at once.

```
// --- message for updating GUI from plugin
bool __stdcall CCustomViewsTwo::checkUpdateGUI(int nControlIndex,
                float fValue, CLinkedList<GUI_PARAMETER>& guiParameters,
                bool bLoadingPreset)
{
    // --- modify GUI using cached pointers in response to GUI changes;
    //      NOTE: must do this here for clients that do not stream zeros
    //      when plugin is idle

    // decode the control index
    switch(nControlIndex)
    {
        case 45: // hide/show button
        {
            // --- always test the pointer first!
            if(m_pVolRightVC)
            {
                // --- note comparison is with fValue
                if(fValue == (float)SWITCH_ON)
                    m_pVolRightVC->setVisible(false);
                else
                    m_pVolRightVC->setVisible(true);
            }
            break;
        }

        // —- etc…
```

Using the cached pointer: moving/resizing GUI elements with *setSize()*
The switch on the GUI that will move this view container is another RackAFX 2-state switch with Control ID 46. In order to move the view container around, you alter the View's *size* member variable. The upper left of this rectangle sets the origin for the control. The width and height set those parameters. In this example, we will move the control down and back up again by 126 pixels. Notice how we hide the control first, then move it, then show it again. Remove the hiding/showing code to see why that needs to be done.

```
case 46: // the Move button
{
        // --- always test the pointer first!
        if(m_pVolRightVC)
        {
                // --- get the size variable from the control
                CRect rect;
                m_pVolRightVC->getViewSize(rect);

                // --- note comparison is with fValue
                if(fValue == (float)SWITCH_ON)
                {
                        // --- move the VC down 20 pixels
                        rect.top += 126;

                        // --- adjust the bottom to accomodate
                        rect.bottom += 126;
                }
                else
                {
                        // --- move the VC up 20 pixels
                        rect.top -= 126;

                        // --- adjust the bottom to accomodate
                        rect.bottom -= 126;
                }

                // --- hide the control
                m_pVolRightVC->setVisible(false);

                // --- set it
                m_pVolRightVC->setViewSize(rect);

                // --- show the control in the new location
                m_pVolRightVC->setVisible(true);

        }
        break;
}
```
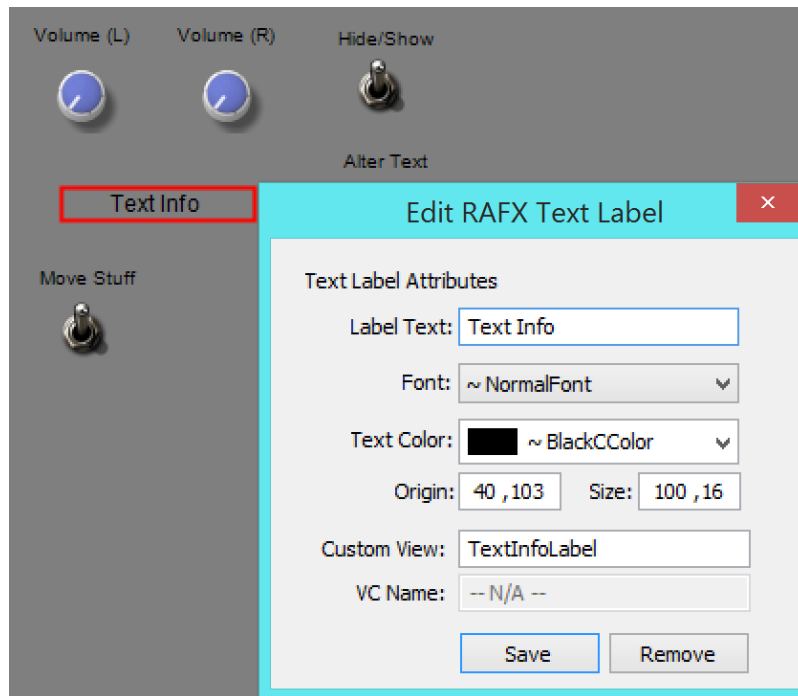
Test the plugin and you can see how these switches work to alter the GUI programmatically. Note that this could also be done from other locations in code, but it is a **BAD IDEA** to place this code in your *processAudio()* methods. The audio thread is high priority and will be called many times.

Example 2: Making a *CTextLabel* Custom View and caching the pointer
Let's continue with a *CTextLabel* example to make a control that whose text can be changed dynamically. This VSTGUI object is actually more complicated that some others because you need to deal with fonts. In addition, we will need to add a call to a cleanup function during the GUI_WILL_CLOSE message. The Custom View name for the text label is TextInfoLabel.

We will need to obey the same rules as usual.

The cached pointer declaration:

```
CTextLabel* m_pTextLabel;
```

The constructor NULLs it out:

```
m_pTextLabel = NULL;
```

We also NULL the pointer in the GUI_WILL_CLOSE message, but we need to call the *cleanup()* method on the internal Font object first:

```
if(m_pTextLabel)
{
      // --- need to take care of the cleanup detail first
      m_pTextLabel->getFont()->cleanup();
      m_pTextLabel = NULL;
}
```

Creation of the object requires setting a Font and Font Color. In both cases you can either user a built-in color or built-in Font, or make your own. Both versions are shown in code. If you are making a font from scratch, you have some control over the style variables. Notice how wire-or-ing is used to set the style.

```
if(info->customViewName.compare("TextInfoLabel") == 0)
{
      // --- get the needed attributes with the helper
      const CRect rect = m_GUIHelper.getRectWithVSTGUIRECT(
                                        info->customViewRect);

      // --- create the label - I am not setting an initial text string
      m_pTextLabel = new CTextLabel(rect);

      // --- make background transparent
      m_pTextLabel->setTransparency(true);

      // --- default colors are defined in ..\vstgui4\vstgui\lib\ccolor.h
      m_pTextLabel->setFontColor(kBlackCColor);

      // --- can also use CColor's static function
      //      (0, 0, 0, 255) is r,g,b,a for BLACK
      // m_pTextLabel->setFontColor(CColor(0, 0, 0, 255));

      // --- FONTS: can use default fonts, or create your own
      //
      // --- Default Font:
      //      default fonts defined in ..\vstgui4\vstgui\lib\cfont.h
      m_pTextLabel->setFont(kNormalFontSmaller);


      // --- return control cloaked as a void*
      return (void*)m_pTextLabel;
}
```

To make a Font "from scratch" you need to do a bit more work:

```
      // --- custom Font:
      //
      // --- construct with Font name (must be installed on system!
      CFontDesc* fontDesc = new CFontDesc("Times New Roman");
      if(fontDesc)
      {
            fontDesc->setSize(10); // points

            // --- set italic: note wire-or-int
            fontDesc->setStyle(fontDesc->getStyle() | kItalicFace);

            // --- set bold: note wire-or-int
            fontDesc->setStyle(fontDesc->getStyle() | kBoldFace);

            // --- can also add underline or strike-through style
            //fontDesc->setStyle(fontDesc->getStyle() | kUnderlineFace);
            //fontDesc->setStyle(fontDesc->getStyle() | kStrikethroughFace);

            m_pTextLabel->setFont(fontDesc);
      }
```

Altering the text
The switch for altering the text has the Control ID of 47. We use the code here to alter the text label, in the *checkUpdateGUI( )* function:

```
    case 47: // alter text
    {
            // --- always test the pointer first!
            if(m_pTextLabel)
            {
                    // --- note comparison is with fValue
                    if(fValue == (float)SWITCH_ON)
                            m_pTextLabel->setText("Alter Text ON");
                    else
                            m_pTextLabel->setText("Alter Text OFF");
            }
            break;
    }
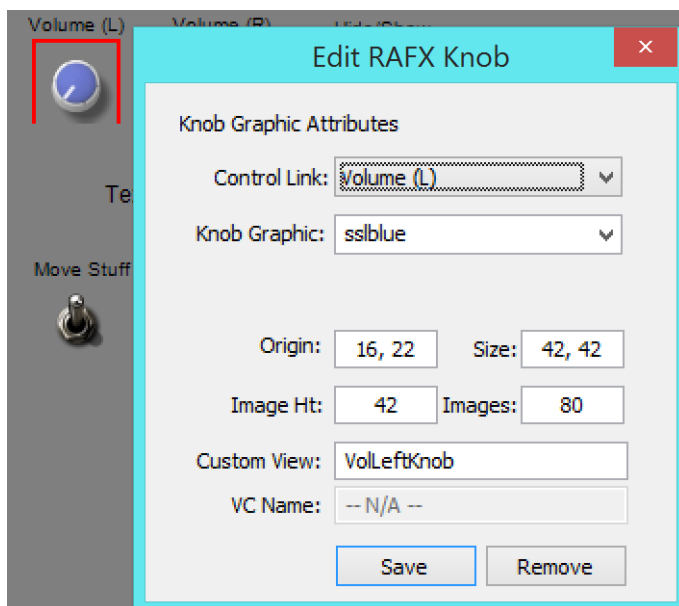```

**Initializing the text**
Ordinarily, you would initialize the text label's string in the constructor. I decided to leave the string blank and initialize it in the GUI_DID_OPEN message handler to show an example of using this message. This also shows the flow of operation - GUI_DID_OPEN is called *after* the Custom View objects have been created!

```
        case GUI_DID_OPEN:
        {
                if(m_pTextLabel)
                        m_pTextLabel->setText("GUI DID OPEN!");

                return NULL;
        }
```

Example 3: Making a *CAnimKnob* Custom View and caching the pointer
Let's finish with a *CAnimKnob* example to show how to change a control's bitmap (graphic) dynamically from within your plugin. Here we will change the graphic for the left volume knob. This is probably a silly example - I am guessing that you might only really want to change graphics for things like view container backgrounds or possibly the meter objects. The Custom View name for the left volume knob is VolLeft-Knob.

We will need to obey the same rules as usual.

The cached pointer declaration:

```
CAnimKnob* m_pLeftVolKnob;
```

The constructor NULLs it out:

```
m_pLeftVolKnob = NULL;
```

We also NULL the pointer in the GUI_WILL_CLOSE message:

```
if(m_pLeftVolKnob)
      m_pLeftVolKnob = NULL;
```

We went over the creation of the *CAnimKnob* object in module 3 so there is no reason to repeat that theory. Here is the code:

```
if(info->customViewName.compare("VolLeftKnob") == 0)
{
      // --- get the needed attributes with the helper
      const CRect rect = m_GUIHelper.getRectWithVSTGUIRECT(
                                      info->customViewRect);
      const CPoint offsetPt = m_GUIHelper.getPointWithVSTGUIPOINT(
                                      info->customViewOffset);
      CBitmap* pBitmap    = m_GUIHelper.loadBitmap(info);

      // --- normal CAnimKnob
      m_pLeftVolKnob = new CAnimKnob(rect,
                              (IControlListener*)info->listener,
                              info->customViewTag,
                              info->customViewSubPixmaps,
                              info->customViewHtOneImage,
                              pBitmap,
                              offsetPt);



      // --- decrement ref count
      if(pBitmap)
            pBitmap->forget();

      // --- return control cloaked as a void*
      return (void*)m_pLeftVolKnob;
}
```

Altering the bitmap
The switch for altering the text has the Control ID of 48. We use the code here to alter the knob's bitmap in the *checkUpdateGUI( )* function. Notice that we still need to call forget() on the new bitmaps, but we do NOT need to do this for the old bitmap - its reference count is decremented when we apply the new bitmap, then it is automatically deleted. Notice that the full bitmap filename is used as the argument for creating the new object. You can find these filename values in the RackAFX.uidesc file's <bitmap> chunk as

we discovered in module 2. **Notice that we always use the m_GUIHelper object's loadBitmap() function to create the bitmap.**

```
case 48: // alter bitmap
{
        // --- always test the pointer first!
        if(m_pLeftVolKnob)
        {
              CBitmap* pBM  = m_pLeftVolKnob->getBackground();

              // --- note comparison is with fValue
              if(fValue == (float)SWITCH_ON)
              {
                     CBitmap* pNewBitmap =
                          m_GUIHelper.loadBitmap("knob303.png");
                     m_pLeftVolKnob->setBackground(pNewBitmap);

                     if(pNewBitmap)
                          pNewBitmap->forget();
              }
              else
              {
                     CBitmap* pNewBitmap =
                          m_GUIHelper.loadBitmap("sslblue.png");
                     m_pLeftVolKnob->setBackground(pNewBitmap);

                     if(pNewBitmap)
                          pNewBitmap->forget();
              }
        }
        break;
}
```

Test the code
Once again, try the project out and make the bitmap change dynamically.

Altering the appearance of Controls
If you want to change a visual aspect of a control, such as its bitmap graphic, colors, etc… at runtime, you need to make these modifications in response to the **GUI_DID_OPEN** message. This is where you can make changes to the initial state of the controls when they first appear on the GUI. For example, I can change the font of the *CTextLabel* and make it italic by responding to this message.

case GUI_DID_OPEN:
{
        if(m_pTextLabel)
        {
                m_pTextLabel->setText("GUI DID OPEN!");

                // --- APPEAARANCE CHANGES ---
                //
                //   Use this for making changes to the control's appearance; starting
                //   with VSTGUI4.3, this must be done during the GUI_DID_OPEN phase.
                //
                //   Examples include changing the bitmap for a control, it's background color, etc...
                //   If customizing the CSegmentButton control, this is where you would create seg
                //   ments, set the text/highlighted-text colors, set the icons (if using), etc...

```
        //
        //    As a simple example, I will alter the font and appearance of the our
        //    CTextLabel control
        //
        // --- construct with Font name (must be installed on system!
        CFontDesc* fontDesc = new CFontDesc("Ariel");
        if(fontDesc)
        {
                fontDesc->setSize(10); // points

                // --- set italic: note wire-or-int
                fontDesc->setStyle(fontDesc->getStyle() | kItalicFace);

                // --- set bold: note wire-or-int
                fontDesc->setStyle(fontDesc->getStyle() | kBoldFace);

                // --- can also add underline or strike-through style
                fontDesc->setStyle(fontDesc->getStyle() | kUnderlineFace);
                //fontDesc->setStyle(fontDesc->getStyle() | kStrikethroughFace);

                // --- make the change
                m_pTextLabel->setFont(fontDesc);
        }
    }
    return NULL;
}
```

Final thoughts

Caching pointers gives you a cool way to change your GUI programmatically from within the plugin but you need to be careful with this. Remember that if you are not careful, you may accidentally try to use a pointer that has no underlying object, so always obey the cached pointer rules. Also remember that you never need to delete VSTGUI objects since they use reference counting - in fact, the destructor is de-clared as *private* to prevent you from even trying!

References:

VSTGUI4 Files and Documentation: http://sourceforge.net/projects/vstgui/