Creating VSTGUI Custom Views 1: **Subclassing VSTGUI4 Objects**

Will Pirkle

VSTGUI4 includes a paradigm known as “custom views” to allow you to further customize the behavior of any VSTGUI control. In RackAFX, it also allows you to create your own **views** (GUI items that display information for the user such as an animation, audio waveform plot, FFT graph, etc…) and **controls** (GUI items that transmit information when the user interacts with them). In this module, we will discuss sub-classing VSTGUI4 controls to change their behavior

In VSTGUI4, RackAFX and these modules, the term “custom view” really means “custom VST-GUI View Object” and that object can be a view-type object or a control-type object.

VSTGUI4 Basics: Views and Controls

VSTGUI4 GUI objects can loosely be classified as *Views* or *Controls*. A View is an object that is visible for the user to see. A View delivers information to the user such as a graph of data, or the location of a knob or slider. You can think of the View part as a one-way flow of information from the GUI to the user’s eyes. A Control is a kind of View because it is also visible to the user and conveys information. However, a Con-trol allows user interaction - for example, the user moves a knob control, and this transmits data back to the plugin - this is a two-way communication path.

VSTGUI4 Objects 1:

CView
CControl

The *CView* object encapsulates the View attributes of a GUI object and the *CControl* object handles the user interaction. In RackAFX there are several *CView* derived objects you can drag and drop into the GUI Designer. These include *CViewContainer*, *UIViewSwitchContainer* and the *CView* object itself.The rest of the GUI objects such as knobs, sliders, buttons, etc… are all derived from *CControl* and *CControl* is de-rived from *CView* so it inherits all the *CView* attributes as well as attributes for generating control informa-tion.

Here are some of the most common attributes and/or methods found on these two basic objects. When you create your own Custom View objects, you typically override a few of these methods - in our exam-ples we will only override two or three functions per objects. In most cases, this is all you will need.

CView

Method	Description
draw()	draws the object with primitive drawing methods
invalid()	re-draws the object as needed, for example to update new information (graph) or as a result of user interaction (control)
setVisible()	shows or hides the control (very powerful function!)

Method	**Description**
onMouseDown()	handle the mouse-down event (for any mouse button)
onMouseUp()	handle the mouse-up event (for any mouse button)
onMouseMoved()	handle the mouse-move event
setBackground()	sets the background bitmap for the view
getBackground()	gets the background bitmap for the object
setViewSize()	sets the location and dimensions of the view object
getViewSize()	returns the coordinates of the view

CControl

Attribute	**Description**
value	a floating point variable that represents the current value of the control
vmin	the minimum value the control can have
vmax	the maximum value the control can have
tag	the integer value that links the control to some underlying variable in the plugin
listener	the object that will receive notifications when the control is adjusted

Method	**Description**
getValue(), setValue()	get/set the current control value
getMin(), setMin()	get/set the minimum control value
getMax(), setMax()	get/set the maximum control value
getTag(), setTag()	get/set the control tag
getListener(), setListener()	get/set the listener object that receives control change notifications
beginEdit()	called when the user begins to adjust the control
endEdit()	called when the user finishes adjusting the control

VSTGUI Conventions for *CControl's value* member
The most important member variable in the *CControl* object is probably the *value* variable which encodes the current value of the GUI control. You can see that the *vmin* and *vmax* limits set the range of the control. The *value* variable can be expressed as either "plain" or "normalized." In almost all VSTGUI controls,

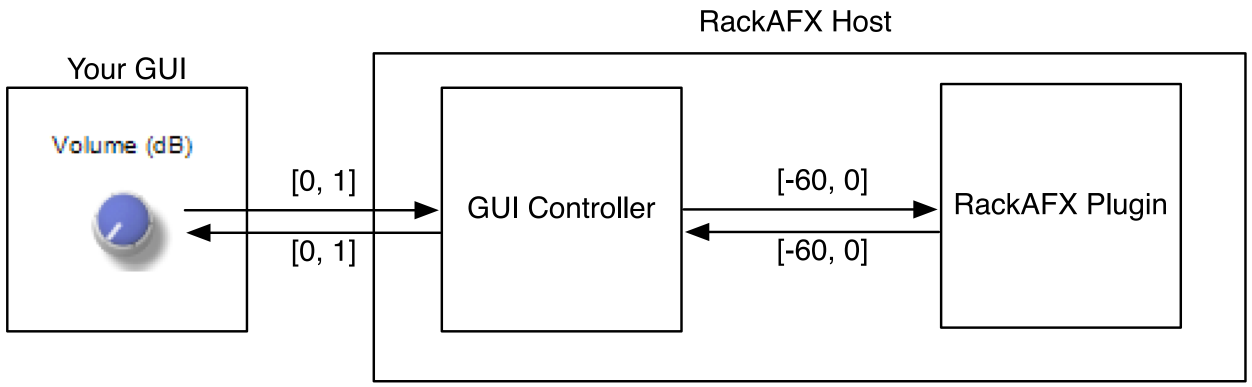
the *value* variable is normalized to a range of 0.0 to 1.0 so the plain and normalized versions are identical.

Since VSTGUI is based on the original VST1.0 API, there is some legacy to deal with here - in the original VST1 and VST2 APIs, all GUI control elements sent and received normalized values to and from the plugin object. When you wrote VST1 and VST2 plugins, you needed to provide functions that converted the normalized values to and from the values that made sense for your plugin object (aka the "plain" values). In VST3 this changed so that most of these details of converting from plain to normalized are handled for you. The newer version of VSTGUI reflects this in that it allows you to set the min and max values to non-normalized limits.

However, the VSTGUI4 objects that are included in the library almost always default to the normalized version in code. So, the knobs, sliders, buttons and the like all transmit/receive values between 0.0 and 1.0. You can change this by subclassing your own versions of the objects!

The three notable exceptions are the *COptionMenu*, *CVerticalSwitch* and *CHorizontalSwitch*, all of which are set to transmit plain values that are used as integer switching values. For example, when you set the value of a COptionMenu to "2" you select the third string in the drop-down list of values (since the control is zero-indexed, value number 2 corresponds to string number 3).

The object that handles the interaction with the controls, converting the plain values to and from the normalized values is called an "*Editor*" in VSTGUI parlance. I don't particularly like this term because we tend to think of an *Editor* as something that is text based. I prefer to call it the "GUI Controller" instead. Let's briefly discuss how the Editor object is implemented in your RackAFX, VST and AU plugin projects. The simplest case is RackAFX:

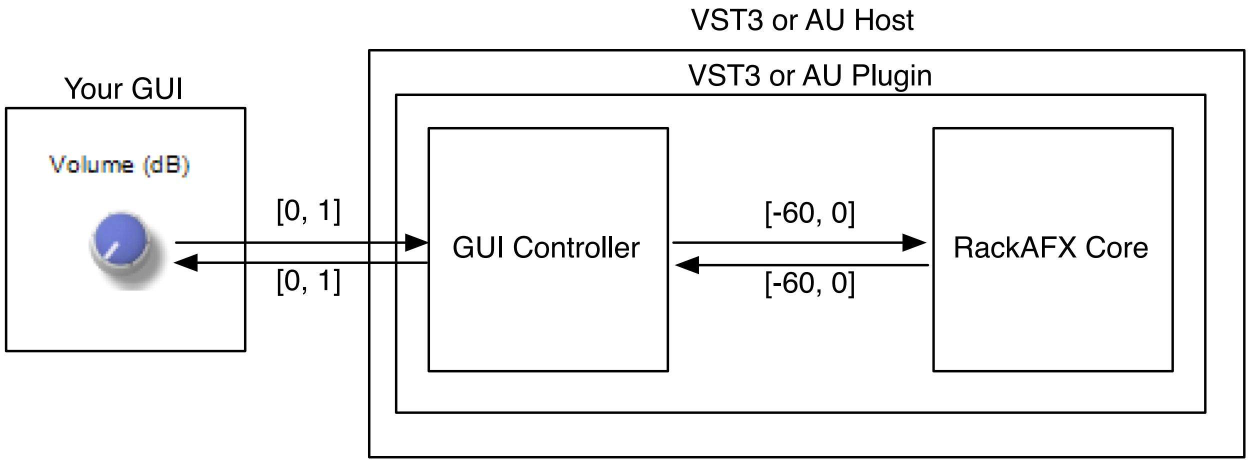


The GUI Controller object is built-into your Plug-In via the static library Sock2VST3. It translates the normalized values into plain values for your RackAFX plugin - this is all completely transparent for you and you do not get to see the GUI Controller object's code.

Remember the *listener* member variable in CControl? The GUI Editor object is the listener and the GUI control is connected to this object via the *listener* member variable.

In VST3 and AU, the ported projects contain a VST2/3 or AU Wrapper object which contains an instance of your RackAFX core (this is also the way most other cross-platform plugin packages operate). The GUI

Editor is contained within the wrapper object. When you use Make VST or Make AU in RackAFX, you can look at the editor object code! The AU version has the most verbose code as some of the details in VST3 are included in the core VST code (which you can also see, but you need to dig deeper to find this code).



Once again, the GUI Controller is the *listener* object. Likewise, since this object is already written for you, it is mostly transparent to your RackAFX code. When you create Custom View objects in the Advanced GUI API, you generally will set the *listener* to the built-in RackAFX/VST/AU editor object. For pure custom GUIs where you write and implement the GUI purely in C++ code, you will need to implement this object. Fortunately, I have already supplied you with an Editor object for each RackAFX project - you will need to fill in the appropriate code. We will get into that in module 6.

The CVSTGUIHelper Object
In the last module, you learned that you need to add three files to any project where you need to implement these advanced GUI concepts. I have written an object for you called *CVSTGUIHelper* which is going to do most of the work for you in extracting the information about the Custom View for you to use. The object is declared and implemented in the two files:

GUIViewAttributes.h
GUIViewAttributes.cpp

To use it, first #include the .h file in your plugin's .h file and then declare an instance of the object in the user variable section of the file. Mine is named

```
#include "plugin.h"
#include "GUIViewAttributes.h"

class CCustomViews : public CPlugIn
{
public:

<SNIP SNIP SNIP>

       // Custom GUI
       virtual void* __stdcall showGUI(void* pInfo);

       // Add your code here: ------------------------------------------- //
       CVSTGUIHelper m_GUIHelper;
```

```
        // END OF USER CODE ---------------------------------------------- //


        etc…
```

We will use this object to glean information about our custom control and we'll go over the functions as the time comes. But, first lets dig a little deeper into the XML file and look at some object descriptions and attributes. The *CVSTGUIHelper* object is going to decode the information contained in the VST-GUI_VIEW_INFO struct that is passed to your *showGUI()* method.

The *CVTSGUIHelper* object is NOT a VST Editor. It is a simple object that helps to convert information from RackAFX (or VST or AU) about your Custom View into meaningful data you can use to set up your Custom View/Control.

Graphics for your Custom Views
Many of the GUI objects require graphics files to display properly. The graphics files are called "bitmaps" in VSTGUI parlance, **however they must be .PNG files for use in your GUIs** since the RAFX-VST libraries expect only PNG file graphics. Fortunately, converting graphic formats is commonplace today.

Graphics from the RackAFX GUI Designer: you automatically have access to all of the graphics files used in the RackAFX GUI Designer. You can find the names of these files easily by inspecting the .uidesc file and navigating to the <bitmaps> chunk.

Graphics from your plugin: you can also add your own graphics to your plugin's resource stream. This is documented on my You Tube video here:

https://www.youtube.com/watch?v=cp3draeYLPU

The process involves 3 steps - if the graphics file is named "knobgraphic.png" then you would do the following:

1. copy the .png file into the <project>/resources folder
2. in the Visual Studio Solution Explorer, find the <project>.rc file and right click on it; choose "View Code" and add a line to your .rc file like this (notice that the all CAPS name is identical to the lower case version):

```
/////////////////////////////////////////////////////////////////////////////
//
// PNG
//
KNOBGRAPHIC.PNG          PNG                    "resources\\knobgraphic.png"
```

3. recompile the RackAFX project - your graphics will now be available in both the RackAFX GUI Designer and also your plugin natively

Always use the *CVTSGUIHelper* object to create bitmaps for your controls. It has two main loadBitmap() functions.

If you are loading the same graphic you assigned in the RackAFX GUI Designer, use:

m_GUIHelper.loadBitmap(info);

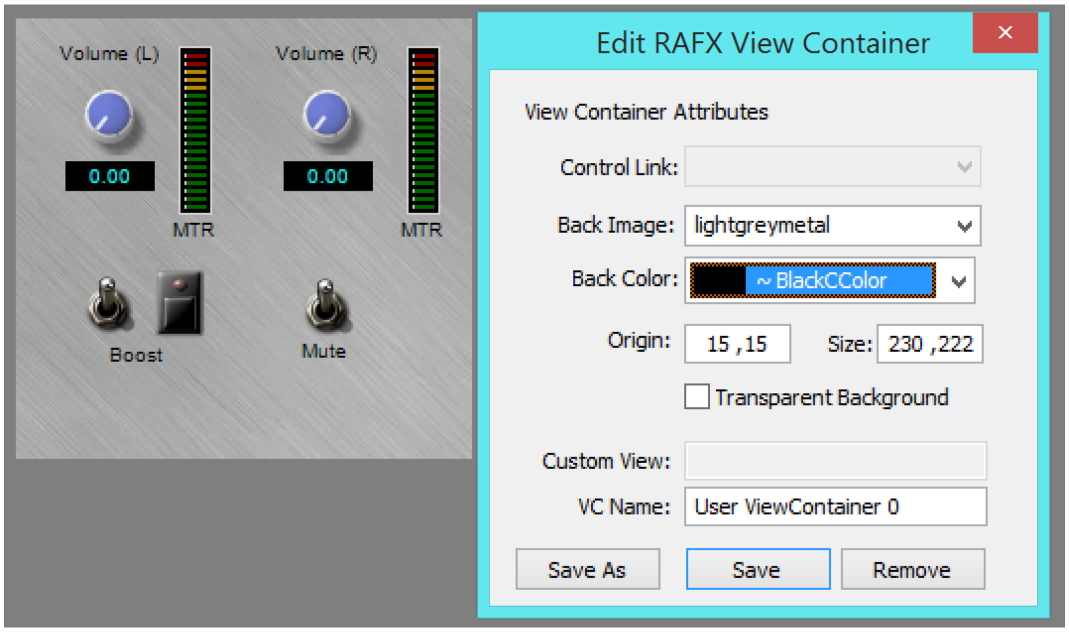
The *info* structure contains the name string of the graphic.

If you are loading your own graphic from your plugin's resources, use the graphic file name:

m_GUIHelper.loadBitmap("knobgraphic.png");

You should use the loadBitmap() functions rather than trying to instantiate the objects with the *new* operator directly. These functions figure out the resource location of your graphic element and load them accordingly.

Getting deeper into the RackAFX.uidesc file
Let's continue poking around the XML file to see how the C++ objects that make up the GUI are described. In the **CustomViews** project, you can see how the CViewContainer is setup. This is what the View Container's attributes look like in RackAFX:



View Container Attributes

Control Link:
Back Image: lightgreymetal
Back Color: ~ BlackCColor
Origin: 15 ,15 Size: 230 ,222
☐ Transparent Background
Custom View:
VC Name: User ViewContainer 0

Notice the Origin and Size attributes. **The Origin is the GUI coordinates (x,y) of the upper left corner of the object relative to it's parent.** Here, the container's origin is offset 15 pixels to the right, and 15 pixels down. Like many GUI coordinate systems, the x-coordinate increases positively as you move to the right, and the y-coordinate increases positively as you move *down*. This flipped coordinate system may be confusing at first, but it is easy to get used to. **The Size is the (width, height) in pixels.**

Next, notice that the VC Name is set to “User ViewContainer 0” — as you learned in the last module, all VSTGUI *CViewContainers* must have unique names. RackAFX generated that unique name, but you are free to change it to something more appropriate for you if you want. Now, examine the XML file and find the chunk near the top (I have wrapped the text string for easier viewing here):

```
<template background-color="~ GreyCColor"
        background-color-draw-style="filled and stroked"
        bitmap=""
        class=“CViewContainer"
        maxSize="265, 257"
        minSize="265, 257"
        mouse-enabled="true"
        name="Editor"
        origin="0, 0"
        size="265, 257"
        transparent=“false">

<view background-color="~ BlackCColor"
        background-color-draw-style="filled and stroked"
        bitmap="lightgreymetal"
        class="CViewContainer"
        mouse-enabled="true"
        origin="15 ,15"
        rafxtemplate-type="userViewContainer"
        size="230 ,222"
        template="User ViewContainer 0"
        transparent="false"
        custom-view-name=""
        sub-controller="" />
</template>
```

The first part marked **<template** is the definition of the outer frame that holds the entire GUI. Check out the attributes:

bitmap=“"	— there is no background bitmap in this GUI
class="CViewContainer"	— this is a CViewContainer object
maxSize="265, 257"	— the GUI is 265 x 257 pixels
minSize="265, 257"	— the GUI is 265 x 257 pixels (min = max = no resizing of the GUI)
name="Editor"	— unique name is Editor
origin="0, 0" size="265, 257"	— origin and size, the main view always has origin = (0,0)

Now look at the second part marked **<view** which describes the single view container that holds the rest of the controls and examine its attributes - you can now get a feel for how RackAFX creates the XML from your drag-and-drop environment. Notice that the unique name is called the “template” for all sub-view containers of the Editor.

template="User ViewContainer 0”

Just after the <control-tag/> chunk, you will find the definition of the View Container - it looks almost like an exact repeat of the line above, except the unique name is now labeled “name:”

```
<template background-color="~ BlackCColor"
        background-color-draw-style="filled and stroked"
        bitmap="lightgreymetal"
        class="CViewContainer"
        mouse-enabled="true"
        name="User ViewContainer 0"
        origin="0, 0"
        size="230 ,222"
        transparent="false">
```

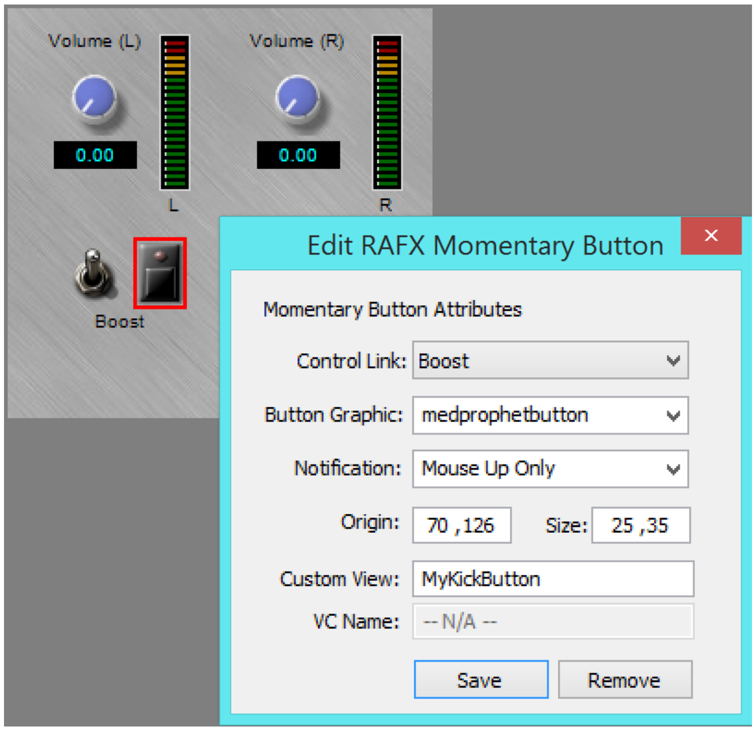
Underneath this, you will find the sub-views of this container. The two knob groups and the two LED meter groups are View Containers and have "template" names, for example:

```
<view background-color=""
        background-color-draw-style="filled and stroked"
        bitmap=""
        class="CViewContainer"
        mouse-enabled="true"
        origin="10 ,5"
        rafxtemplate-type="knobgroup"
        size="75 ,82"
        template="Rafx KnobGroup 0"
        transparent="true"
        custom-view-name=""
        sub-controller="" />
```

Scrolling down a bit, you will find the **Rafx KnobGroup 0** definition with its sub-views. This template/name paradigm is how containers can be embedded.

Now, take a look inside the **User ViewContainer 0** chunk and you will find the code for the momentary button that is connected to the Boost variable - here are its first few attributes:

```
background-offset="0, 0"
bitmap="medprophetbutton"
class="CKickButton"
control-tag="Boost"
custom-view-name="MyKickButton"
```



Momentary Button Attributes

Control Link: Boost
Button Graphic: medprophetbutton
Notification: Mouse Up Only
Origin: 70 ,126
Size: 25 ,35
Custom View: MyKickButton
VC Name: -- N/A --

The VSTGUI object named *CKickButton* is the momentary on-off button. In RackAFX, the only buttons like this are the assignable buttons B1-B3. However, you can still assign momentary buttons to any RackAFX variable. In this module, we are going to subclass the *CKickButton* object to change its behavior. I changed the custom view to *MyKickButton*.

RackAFX also subclasses the *CKickButton* and we'll discuss that after we subclass it for ourselves. On the left is the Momentary Button setup dialog you see in the GUI Designer

You can ignore the field marked "Notification" for now - it is part of the RackAFX subclassed *CKickButton* and will make sense after completing this module. The main things to note here are the custom view name and the origin and size attributes.

VSTGUI4 Objects 2:

CCoord
CPoint
CRect
CBitmap

Its time to get our hands dirty with some more VSTGUI4 objects. These four objects are important since they are used in almost every VSTGUI control. Most of the control GUI's require all four of these objects to be passed into the Constructor so you should get to know them well.

CCoord is not actually an object, it is simply a redefinition of the double datatype:

```
typedef double CCoord;  ///< coordinate type
```

You can essentially use it interchangeably with both the *float* and *double* datatypes.

CPoint encapsulates the concept of a (x,y) *point* or a (width,height) *size*. It uses a union of *CCoords* to create two sets of variables that encode the same information - just use whichever version is most comfortable. Since the object is designed to model a point or a size, the variables are named accordingly:

CCoord x = x-coordinate
CCoord y = y-coordinate

CCoord v = width = x
CCoord h = height = y

Example:

```
CPoint point; // declare

point.x = 10.0; // this also sets the v variable to 10.0
point.y = 20.0; // this also sets the h variable to 20.0

float width = point.v; // get value as a width and float
double height = point.h; // get value as a height and double
```

CRect encapsulates the concept of a rectangle. All VSTGUI controls and views are described with a *CRect* that positions it on the screen and sets the size. The *CRect* object has the following variables and methods (this is not the complete list, just the most important):

Member Variables: these specify the x and y coordinates for the four sides of the rectangles.

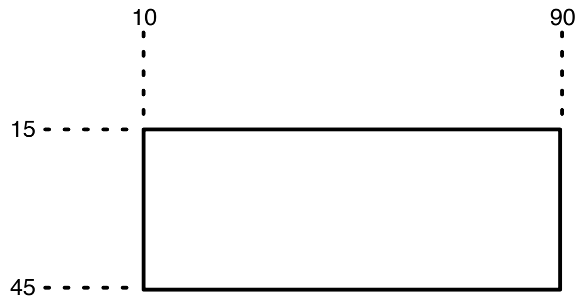
CCoord left;
CCoord right;
CCoord top;
CCoord bottom;

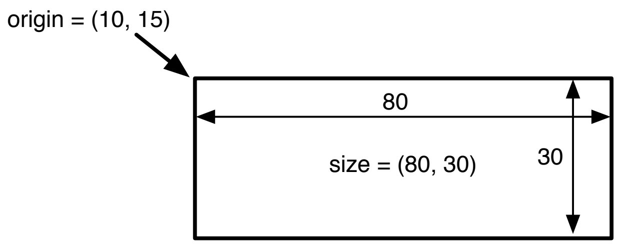
Member Methods: these are self explanatory; see the crect.h file for a bunch more of the simple and helpful functions.

inline CCoord getWidth () const { return right - left; }
inline CCoord getHeight () const { return bottom - top; }

inline void setWidth (CCoord width) { right = left + width; }
inline void setHeight (CCoord height) { bottom = top + height; }

There are two normal constructors and one copy-constructor for the object. Here is a rectangle with the four sides labeled as well as the origin/size attributes:





The most straightforward way to define the rect is by supplying the four coordinates left, right, top and bottom, in the constructor.

```
CRect rect(10, 15, 90, 45); // define a rect with (left, top, right, bottom)
```

You can also construct a *CRect* with the origin and size attributes as *CPoints*:

```
CPoint origin(10,15);
CPoint size(80,30);

CRect rect(origin, size);
```

CBitmap encapsulates a graphic that is usually constructed with a graphic file name or what VSTGUI4 calls a *CResourceDescription*. The bitmap is constructed and "loaded" and the object is passed to the Custom View's constructor. The *CVSTGUIHelper* object will do all of this work for you.

Reference counting: the *forget()* method and CBitmaps
VSTGUI4 uses reference counting to automatically delete objects that are no longer needed. If you are new to reference counting, you might want to find a website that explains the basics. Most of the time, we don't need to worry about reference counting. But, when we use a *CBitmap* object as part of the creation of a custom view object, its reference count is incremented during the view object's construction. So, after

we create a custom view object, we will need to decrement the reference count on the CBitmap object(s) that we used. The decrement function is called *forget()*. You will see a repeated pattern like this:

- create the *CBitmap* object
- use the object to create a *CView* object
- call *forget()* on the *CBitmap* object

VSTGUI4 Objects 3:

CControlListener

The *CControlListener* object is critical to VSTGUI's functionality. This object is not a view or a control. It is the object that will be receiving notifications whenever the user adjusts the control. There are two possibilities here:

1. The VST Editor object is usually the *CControlListener* - this is a special object in RackAFX, VST and AU that is setup to get notifications and then set your plugin's parameters accordingly and described above.
2. If you implement an advanced VSTGUI control — either a control that RackAFX does not support like *CCheckBox*, or a custom control you design — then you need to supply an object that will act as the *CControlListener* and which will alter your plugin's variables. We will do an example like this later.

Using *showGUI()* to decode the custom object name
When the GUI loads, the framework will call your plugin's *showGUI()* method repeatedly for each control with a custom view. The name of the custom view is transmitted inside the **VSTGUI_VIEW_INFO** structure that is passed into *showGUI()* as an argument. In the last module, we decoded the *message* variable and in this module, we will decode the *customViewName* attribute so that we can create our custom view. The **VSTGUI_VIEW_INFO** structure has a bunch of variables that are all related to the custom view. Let's examine the attributes. You can find the declaration of the structure in the RAFXViewStructures.h file.

```
// --- custom view stuff
string   customViewName;
```

We will decode the *customViewName* variable. Since we are using std::string, we can use the *compare()* method to compare the string against *MyKickButton*. The *showGUI()* method will now look like this:

```
void* __stdcall CCustomViews::showGUI(void* pInfo)
{
      // --- ALWAYS try base class first in case of future updates
      void* result = CPlugIn::showGUI(pInfo);
      if(result)
            return result;

      // --- uncloak the info struct
      VSTGUI_VIEW_INFO* info = (VSTGUI_VIEW_INFO*)pInfo;
      if(!info) return NULL;

      switch(info->message)
      {
            <SNIP SNIP SNIP>

            case GUI_CUSTOMVIEW:
```

```
{
      // --- decode the customViewName variable
      if(info->customViewName.compare("MyKickButton") == 0)
      {
            // create the control

            // return the object pointer, cloaked as void*

      }

      return NULL; //
}

etc…
```

EXAMPLE: Subclassing the CKickButton object to override mouse-down and mouse-up behavior

CKickButton

The *CKickButton* object encapsulates a momentary on/off button or switch. It uses a graphic (bitmap) file that has both states of the switch in it and then it shows and hides only the part of the image with the appropriate button state as you select the button. For demonstration purposes, we are going to first create a normal *CKickButton* object as our custom view. Then, we'll subclass the object and modify it.

The first thing to do is locate the .h file for the *CKickButton* object. Most of the GUI control objects are located in the *controls* subfolder. At the top of the CustomViews.cpp file, you can see the #include statement:

```
#include "../vstgui4/vstgui/lib/controls/cbuttons.h"
```

Notice the relative path, denoted by "../" which means "in a folder parallel to this one" which is why you install the vstgui4 folder in the same folder as your RackAFX projects. If you need to have the vstgui4 folder in some other location, you need to alter this relative path. Right click on the #include line and choose "Open Document …" and find the *CKickButton* class definition and constructors:

We are going to use the first constructor as our preferred method:

```
CKickButton(const CRect&      size,
            CControlListener* listener,
            int32_t           tag,
            CBitmap*          background,
            const CPoint&     offset = CPoint (0, 0));
```

The arguments are:

CRect size	the rectangle that describes the location and dimensions of the control
CControlListener*	the listener object
int32_t tag	the integer tag value
CBitmap* background	the image for the control
CPoint offset	the x,y offset location of the upper left corner of the control, usually 0,0

All of this information you need to construct the object arrives packaged inside the VSTGUI_VIEW_INFO structure that is passed into your *showGU()* method. We will use the *CVSTGUIHelper* to extract the information. The best way to learn is by example. Look at the VSTGUI_VIEW_INFO definition to see the variables we will use for the *CKickButton* example:

```
int           customViewTag;
VSTGUIRECT    customViewRect;
VSTGUIPOINT customViewOffset;
string        customViewBitmapName;
void*         listener;
```

Notice that the *CControlListener* pointer is passed as a void* — like the info structure, you will need to uncloak it to use it. Here is how you use the helper object to extract the information, then use the constructor to make the object:

```
// --- decode the customViewName variable
if(info->customViewName.compare("MyKickButton") == 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);

      // --- create it!
      CKickButton* pButton = new CKickButton(rect,
                                             (CControlListener*)info->listener,
                                              info->customViewTag,
                                              pBitmap,
                                              offsetPt);

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


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

Notice the *forget()* function to decrement the bitmap's reference count. Compile the code with the standard *CKickButton* as above (the sample code has both normal and subclassed versions). This button is connected to the *m_uBoost* variable that is either 0 (SWITCH_OFF) or 1 (SWITCH_ON). Ideally, we would like a button that behaves like this:

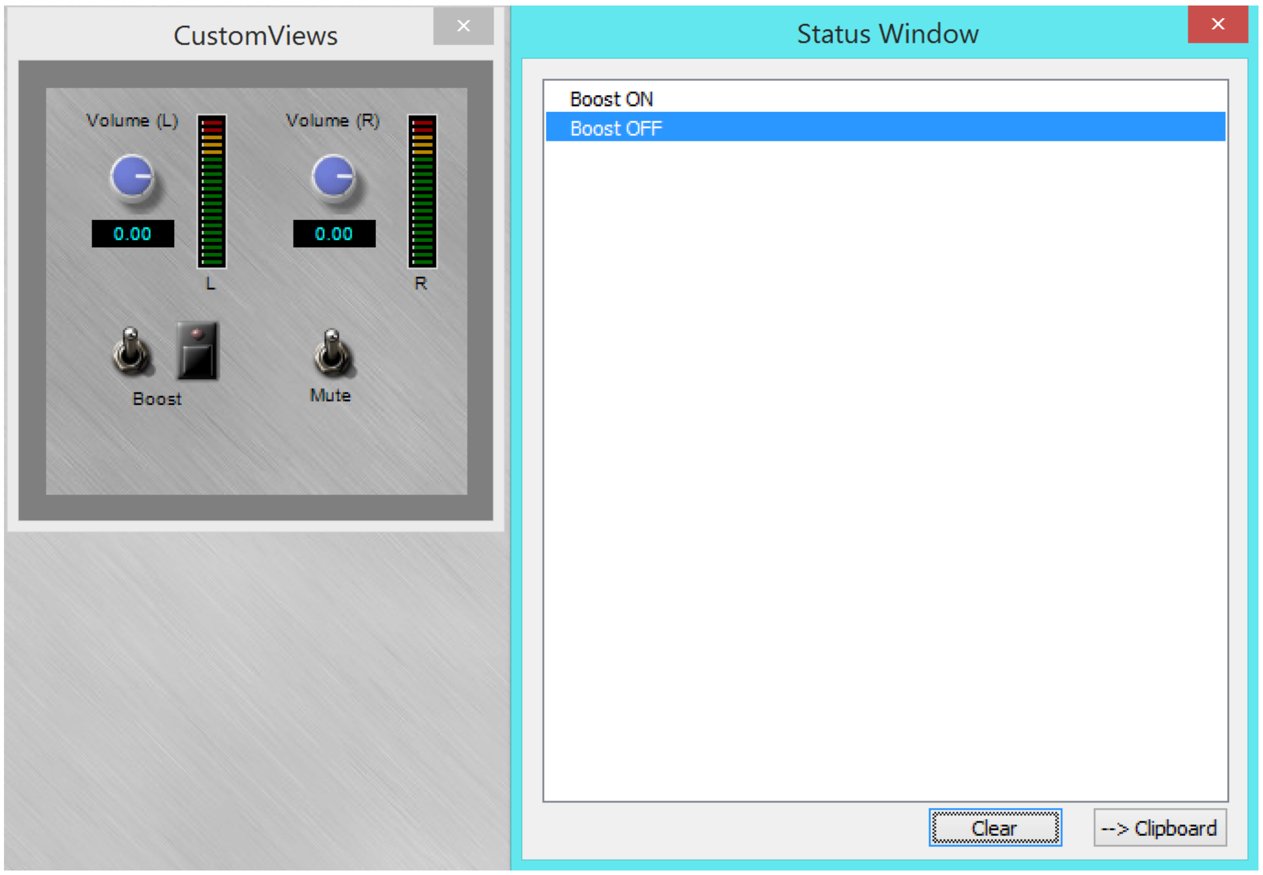
press button and hold it:	m_uBoost = 1
release button:	m_uBoost = 0

However, when you load the plugin and try the control, nothing happens! The control's LED light does not light up and there is no Boost effect (which amplifies the output by 10X — very audible). What's going on here? One way to figure it out is to use the *sendStatusWndText()* function in RackAFX to send some debug text to the Status Window when the *userInterfaceChange()* function is called in response to the GUI button presses. Here is the code:

```
bool __stdcall CCustomViews::userInterfaceChange(int nControlIndex)
```

```
{
        switch(nControlIndex)
        {
                case 45: // 45 is boost control
                {
                        if(m_uBoost == 1)
                                sendStatusWndText("Boost ON");
                        if(m_uBoost == 0)
                                sendStatusWndText("Boost OFF");

                        break;
                }

                default:
                        break;
        }

        return true;
}
```

Now, run the sample project again with the status window open and press the boost button. The status windows shows what is happening:



```
Boost ON
Boost OFF
```

When you press the *CKickButton*, you get an ON (1) signal immediately followed by an OFF (0) signal. This is the default behavior of the *CKickButton* in VSTGUI4. We can modify the behavior by subclassing the control and changing the two functions that handle the mouse-down and mouse-up events. We can modify it in multiple ways:

- transmit an ON signal when the button is held down, and an OFF signal when the button is released (this is what we want)
- transmit a notification on the mouse-down event only
- transmit a notification on the mouse-up event only*

(*) this is the way the Assignable Buttons B1, B2 and B3 work in RackAFX when not in latching mode!

To make this Boost button work properly, we need to modify the control to give the ON/OFF notifications in the first way above. To do that we will subclass the control.

Subclassing *CKickButton*
Subclassing a VSTGUI object is no different from subclassing any other C++ object - you create a derived class and override/alter any functions you need to. For this example, we need to alter the two functions that get called when the mouse is down or up. It is important to make your subclassed object names as unique as you can to avoid name collisions. I have added my initials WCP to the end of the class name. In Visual Studio, create a new Class (right-click on the Solution and choose *Add->Class*) and give it CK-ickButton as the base. Here are some things to note:

- VSTGUI4 is namespaced
- in many (but not all) virtual functions, there is a macro for C++11 support called VSTGUI_OVER-RIDE_VMETHOD which is simply tacked onto the end of the function

```
#ifndef __WPKICKBUTTON__
#define __WPKICKBUTTON__
#include "../vstgui4/vstgui/lib/controls/cbuttons.h"

/* -------------------------------------------------------
     CKickButtonWCP
     Custom VSTGUI Object by Will Pirkle
     Created with RackAFX(TM) Plugin Development Software
     www.willpirkle.com
 -------------------------------------------------------*/
namespace VSTGUI {

class CKickButtonWCP : public CKickButton
{
public:
      // - constructor
      CKickButtonWCP(const CRect& size,
                     CControlListener* listener,
                     int32_t tag,
                     CBitmap* background,
                     const CPoint& offset = CPoint (0, 0));

      // - mouse down override
      virtual CMouseEventResult onMouseDown(CPoint& where,
                                            const CButtonState& buttons)
                                            VSTGUI_OVERRIDE_VMETHOD;

      // - mouse up override
      virtual CMouseEventResult onMouseUp(CPoint& where,
                                          const CButtonState& buttons)
```

```
                                                VSTGUI_OVERRIDE_VMETHOD;

private:
      float   fEntryState;
};

}
```

I will leave it up to you to study the way VSTGUI works as far as the value changes occur - you are going to need to step into the code and get a little dirty if you want to get good at VSTGUI programming. To make life simple, we will just cut and paste the original *onMouseDown()* and *onMouseUp()* code from the *CKickButton*, and then modify that. The constructor does nothing but call the base class so it is not shown here, but you can examine it in the sample code. The only modification that I made was to comment out a few lines of code. You can see that the function *valueChanged()* gets called twice, once with the max value (1) and again with the min value (0) which is accessed with the *getMin()* function. This is what is causing the immediate ON/OFF behavior of the control.

In all *CControl* based objects, the editing/changing of the control's underlying *value* member is framed with two function calls, *beginEdit()* and *endEdit().* You can see that both of these functions are called in the mouse down method, and this is causing the button's graphic to not appear to switch states.

The first part of the fix is to comment out the code that resets the value variable to the minimum, and the code that calls *valueChanged()* a second time. The second part of the fix is to comment out the *endEdit()* code.

```
VSTGUI::CMouseEventResult CKickButtonWCP::onMouseDown(CPoint& where,
                                                    const CButtonState& buttons)
{
      if (!(buttons & kLButton))
            return kMouseEventNotHandled;

      value = 1.0;
      fEntryState = value;

      // start the edit/change
      beginEdit();

      if (value)
            valueChanged ();
      //value = getMin ();
      // valueChanged ();
      if (isDirty ())
            invalid ();
      //endEdit ();

      return onMouseMoved (where, buttons);
}
```

We also need to alter the mouse-up handler. First, another redundant *valueChange()* method is commented out, and I've added the endEdit() function call to complete the edit/change operation after the mouse button is released.

```
VSTGUI::CMouseEventResult CKickButtonWCP::onMouseUp(CPoint& where,
                                                    const CButtonState& buttons)
{
      //if (value)
      //    valueChanged ();
      value = getMin ();
      valueChanged ();
      if (isDirty ())
            invalid ();
      // added this
      endEdit ();

      return kMouseEventHandled;
}
```

Using this new C++ object as the custom control is simple since it uses the same constructor we implemented with *CKickButton*. You add the necessary #include statement for the new object file and then use the new constructor. At the top of the .cpp file:

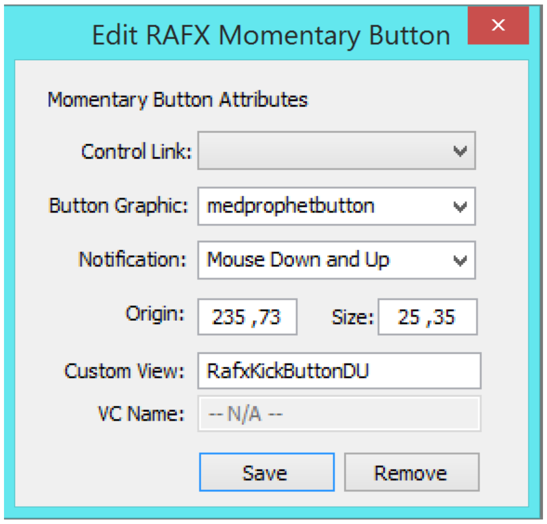
```
#include "CustomViews.h"
#include "../vstgui4/vstgui/lib/controls/cbuttons.h" // normal CKickButton
#include "KickButtonWCP.h"    // for new subclassed control
```

And, then just use the constructor in the *showGUI()* method:

```
// --- 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);

// --- create it!
CKickButtonWCP* pButton = new CKickButtonWCP(rect,
                                            (CControlListener*)info->listener,
                                             info->customViewTag,
                                             pBitmap,
                                             offsetPt);

// — ordinary CKickButton
//CKickButton* pButton = new CKickButton(rect,
//                                          (CControlListener*)info->listener,
//                                           info->customViewTag,
//                                           pBitmap,
//                                           offsetPt);

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

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

Now, compile the code again and notice the behavior change - when you hold the BOOST button down, the control engages and when you release it, it disengages.

***CKickButton* in the RackAFX GUI Designer**



Momentary Button Attributes

Control Link:

Button Graphic: medprophetbutton

Notification: Mouse Down and Up

Origin: 235 ,73 Size: 25 ,35

Custom View: RafxKickButtonDU

VC Name: -- N/A --

Because of the mouse issue with *CKickButton*, this was one of the first controls I subclassed in RackAFX. In my subclassed version, you can select how you want the notifications to occur - on mouse-up and down, or just mouse-down or mouse-up only. When you drag a new *CKickButton* control into the GUI designer, the attribute dialog looks like what you see on the left. You choose how you want the notifications to occur with the Notification field. The Custom View name is used to decode the information. Here it is *RafxKickButtonDU* where DU stands for Down and Up. Note that as you change the Notification field, the Custom Name changes accordingly. The name encodes the custom behavior that is passed to the control. You can see the details of my custom object in both the VST and AU ported projects. You can also find the code that decodes the string and sets the variable on the object.

Sliver Challenge:

Modify the *CKickButtonWCP* object (or better yet, create your own subclass) that mimics the custom control in RackAFX and allows you to choose the kind of mouse behavior you want - mouse down/up, mouse down or mouse up. You might also want to include the original *CKickButton* mouse implementation as the "classic" style. Encode the mouse behavior in the Custom View name string.

EXAMPLE: Subclassing the CAnimKnob object to override the draw() method

CAnimKnob

The *CAnimKnob* object encapsulates a knob control that is "animated" — this is a more advanced version of the original VSTGUI control called *CKnob*. The animated knob uses a graphic that is arranged like a film-strip — a vertical set of images that represent the knob in all its different positions. When you move the knob, the *draw()* method crops the graphic to show you just the single image that represents the control in one position. In this example, we are not going to mess with the cropping code. Instead, we are going to alter the *value* variable to make the knob move in reverse. So, when you drag the mouse up, the knob will turn in the counterclockwise direction. This is not particularly useful on its own, but it will get you started on the *draw()* method so you can try to implement the Challenge at the end of this section.

Locate the .h file for the *CAnimKnob* object. At the top of the CustomViews.cpp file, you can see the #include statement:

```
#include "vstgui/lib/controls/cknob.h"
```

We are going to use the second constructor as our preferred method:

```
CAnimKnob(const CRect&         size,
          CControlListener*    listener,
          int32_t              tag,
          int32_t              subPixmaps,
          CCoord               heightOfOneImage,
          CBitmap*           background,
          const CPoint&      offset = CPoint (0, 0));
```

The arguments are:

CRect size	the rectangle that describes the location and dimensions of the control
CControlListener*	the listener object
int32_t tag	the integer tag value
int32_t subPixMaps	the number of animation frames in the knob graphic
CCoord heightOfOneImage	the height of one image “frame” in the animation strip
CBitmap* background	the image for the control
CPoint offset	the x,y offset location of the upper left corner of the control, usually 0,0

My derived class is named *CKnobWCP* and is packaged in the *KnobWCP.h* and *KnobWCP.cpp* files. We will make our derived constructor follow the ordinary object and we override the *draw()* method to change the way the control draws. Here is my subclassed control definition:

```
#ifndef __cknobwcp__
#define __cknobwcp__

#include "../vstgui4/vstgui/lib/controls/cknob.h"

/* -------------------------------------------------------
     CKnobWCP
     Custom VSTGUI Object by Will Pirkle
     Created with RackAFX(TM) Plugin Development Software
     www.willpirkle.com
 -------------------------------------------------------*/

namespace VSTGUI {

class CKnobWCP : public CAnimKnob
{
public:
      // --- constructor
      CKnobWCP(const CRect& size, CControlListener* listener,
               int32_t tag, int32_t subPixmaps, CCoord heightOfOneImage,
               CBitmap* background, const CPoint& offset = CPoint (0, 0));

      // --- one function to override
      virtual void draw (CDrawContext* pContext) VSTGUI_OVERRIDE_VMETHOD;
};

}

#endif
```

First, have a look at the normal drawing code for *CAnimKnob* (again, you need to examine the code and figure out the basic way the function works on your own). The argument is a *CDrawContext* which is the platform independent drawing object that renders the graphic. We will discuss this object more in module 5 when we draw our own graphics.

The code that actually renders the graphic is:

```
getDrawBackground()->draw(pContext, getViewSize (), where);
```

getDrawBackground() returns a pointer to the *CBitmap* object that is the animation strip. The *CBitmap's draw()* function is the code that renders the graphic into the device context.

You can see that the logic uses the *value* variable in several locations (in **bold**). This code sets up a *CPoint* variable named *where* that is used in the bitmap's draw function below it:

```
getDrawBackground()->draw(pContext, getViewSize (), where);
```

This *CPoint* object *where* tells the draw function where to clip the graphic.

```
void CAnimKnob::draw(CDrawContext *pContext)
{
	if(getDrawBackground())
	{
		CPoint where (0, 0);
		if (value >= 0.f && heightOfOneImage > 0.)
		{
			CCoord tmp = heightOfOneImage * (getNumSubPixmaps () - 1);
			if (bInverseBitmap)
				where.v = floor ((1. - value) * tmp);
			else
				where.v = floor (value * tmp);
			where.v -= (int32_t)where.v % (int32_t)heightOfOneImage;
		}

		getDrawBackground()->draw(pContext, getViewSize (), where);
	}
	setDirty (false);
}
```

If we want to invert the behavior of the control there are two options:

- invert the *value* variable and reverse it to the range [1, 0]
- subclass the CBitmap control, override its *draw()* function, and alter the code to reverse the location of the *where* variable

The first option is the simplest and actually the cleanest option. If we subclassed *CBitmap* to *CBitmapWCP*, then we would need to do more work, replacing the *CAnimKnob's* bitmap object with our new one. Since we are going to do some custom drawing of our own in module 6, we will just alter the *value* variable. The code to invert the value is simple and straightforward. However you will note that I first save the original *value* as *tempValue* and then restore it at the end of the function. You could simply implement the reverse logic to invert the *value* back to its original. However, I found that in overriding other controls over time, I've learned to use the save/restore paradigm. In some cases, I do not alter the original *value*, but instead use the tempValue variable. There are reasons for doing this in some cases. Here it is cleaner to change the *value* variable directly.

Here is the subclassed method:

```
void CKnobWCP::draw(CDrawContext *pContext)
{
	if(getDrawBackground ())
	{
		// --- in order to make a reverse knob,
		//     need to alter the value variable
		// --- invert so that 0->1 and 1->0
		double tempValue = value; // save it
```

```
          // --- invert so that 0->1 and 1->0
          value = -value + 1.0;

          CPoint where (0, 0);
          if(value >= 0.f && heightOfOneImage > 0.)
          {
                CCoord tmp = heightOfOneImage * (getNumSubPixmaps () - 1);
                if (bInverseBitmap)
                      where.v = floor ((1. - value) * tmp);
                else
                      where.v = floor (value * tmp);
                where.v -= (int32_t)where.v % (int32_t)heightOfOneImage;
          }

          getDrawBackground()->draw(pContext, getViewSize(), where);

          // --- restore old value
          value = tempValue;
    }
    setDirty (false);
}
```

And here is the instantiation - identical to the normal constructor:

```
    // --- custom reverse version
    CKnobWCP* pKnob = new CKnobWCP(rect,
                                   (CControlListener*)info->listener,
                                    info->customViewTag,
                                    info->customViewSubPixmaps,
                                    info->customViewHtOneImage,
                                    pBitmap,
                                    offsetPt);

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

Compile the code with this new object (just uncomment and re-comment the code as needed) and watch what happens with the control. It initializes itself to the inverted (opposite) location. And, as you move the mouse up and down, it moves in the opposite direction. Notice also that the data in the Edit control (below the knob) matches the location of the control — it is also opposite of the normal knob. The reason this works so easily is that the Edit control is connected to the same **control-tag** as the knob. In a sense, they then share the same *value* variable. So, the Edit readout matches without us needing to make a “reverse edit control” or other nonsense.

Object Destruction?

You might notice that there is no code to destroy our custom views. The good news is that we don’t need to worry about destruction! The reason is that VSTGUI4 uses reference counting to delete our objects when they are no longer needed.

Gold Challenge:

In the CustomViews project that accompanies this module, you will find many more custom view object decoding and instantiation below the *CKickButton* and *CAnimKnob* cases in the logic. In these examples, I show you how to:

- decode more information from the VSTGUI_VIEW_INFO using the *CVSTHelper* object
- instantiate different controls, all are simply normal VSTGUI4 objects

The following objects are included:

COnOffButton
CVerticalSlider
COptionMenu
CVuMeter

Why give us examples of Custom Views of regular controls?
There are three reasons for creating "regular" custom controls. First, each of these examples shows more customization possibilities as well as new items in the VSTGUI_VIEW_INFO struct. You need to play with the CustomViews project and in some cases add some more controls, then set their Custom Names to the values in the function (or whatever you want - it's up to you to decode them). Doing these examples will help you learn and understand.

Second, in module 4 you will learn how to cache the custom view pointers and use them to modify the GUI programmatically from within your plugin. This allows you to make simple adjustments to the GUI if needed, such as showing/hiding objects, changing text, etc…

Third, in module 5 we will use Custom Views to implement objects that are not included in the RackAFX GUI designer. Examples include the check-box and text-button controls as well as advanced views like the Open GL, splitter, movie and gradient views.

Once you can instantiate these objects and use them, try to think of other customizations you might like. Here are some details:

COnOffButton
There are already 2 COnOffButton objects in the UI, one toggle switch for BOOST and the other toggle for MUTE. Make a Custom View for one or both. This one is easy - the arguments for the constructor are identical to the *CKickButton*.

CVerticalSlider
You will need to drag a Vertical Slider control into the GUI Designer and connect it to one of the Volume controls, then give it the correct Custom View name as in the code. This object introduces another *CBitmap* — it requires two bitmaps, one for the slider "groove" and the other for the slider "handle" (also called the "paddle" or "thumb"). Several of the VSTGUI objects require this second bitmap. The example code shows you where to find it in the VSTGUI_VIEW_INFO structure. The *CVerticalSlider* (and *CHorizontalSlider*) both require that you input the min and max in the constructor - for normal controls, they will be 0.0 and 1.0 respectively.

COptionMenu
You will need to drag an Option Menu control into the GUI Designer and connect it to one of the Volume controls, then give it the correct Custom View name as in the codeThis example shows several more attributes that can be set; this object shares attributes with *CTextLabel* and *CTextEdit* controls. These controls allow you to place a frame around them if you want. By examining the way this control is constructed and its attributes set, you can also make your own *CTextLabel* and *CTextEdit* subclasses.

These controls are also color-intensive and have several different color attributes. In VSTGUI4, the *CColor* object represents a color consisting of red, green, blue, and alpha values.

In addition, there are several *styles* available for this family of controls and the example code shows you how to wire-or these style-constants into a single *style* variable. In this control, you can dictate a rounded-rect style where the corners are round and the object is oval shaped as well as a no-frame style that omits the frame component.

CVuMeter
The last object is a VU meter; you can set the Custom View name for one of the two meters on the GUI to play with this control. Like the slider, it requires two graphics, on for the LED on-state and the other for the OFF state. The drawing code figures out where to crop each bitmap so that the LED is partially on/off depending on the current value. Interestingly, this control has no control-tag variable in its constructor so you need to set it manually with *setTag()*.

Here are the rest of the items in the VSTGUI_VIEW_INFO structure that you will learn about by adding your own Custom View names in RackAFX, then watching how they are constructed in the plugin's *showGUI()* function.

```
// --- custom view stuff
char*           customViewName;
int             customViewTag;
VSTGUIRECT      customViewRect;
VSTGUIPOINT customViewOffset;
char*           customViewBitmapName;
char*           customViewHandleBitmapName;         // sliders
char*           customViewOffBitmapName;            // LED Meters
char*           customViewOrientation;              // sliders, switches, meters

void*           customViewBackColor;        // CColor cloaked
void*           customViewFrameColor;       // CColor cloaked
void*           customViewFontColor;        // CColor cloaked
int             customViewFrameWidth;
int             customViewRoundRectRadius;
bool            customViewStyleNoFrame;
bool            customViewStyleRoundRect;

int             customViewHtOneImage;       // CAnimKnob
int             customViewSubPixmaps;       // CAnimKnob

// --- 9-part tiled offsets
bool            isNinePartTiledBitmap;
double          nptoLeft;
double          nptoTop;
double          nptoRight;
double          nptoBottom;
```

In module we covered:
- how VSTGUI Views and Controls work
- normalized versus plain GUI values
- how to poke around in the *RackAFX.uidesc* file to see how C++ objects are described in XML

- decoding the *customViewName* member of the **VSTGUI_VIEW_INFO** structure in the *showGUI()* method
- instantiating several VSTGUI objects
- creating Custom Views in your code with normal VSTGUI objects
- creating Custom Views in your code with subclassed VSTGUI objects
- overriding mouse behavior
- overriding the *draw()* function

References:

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