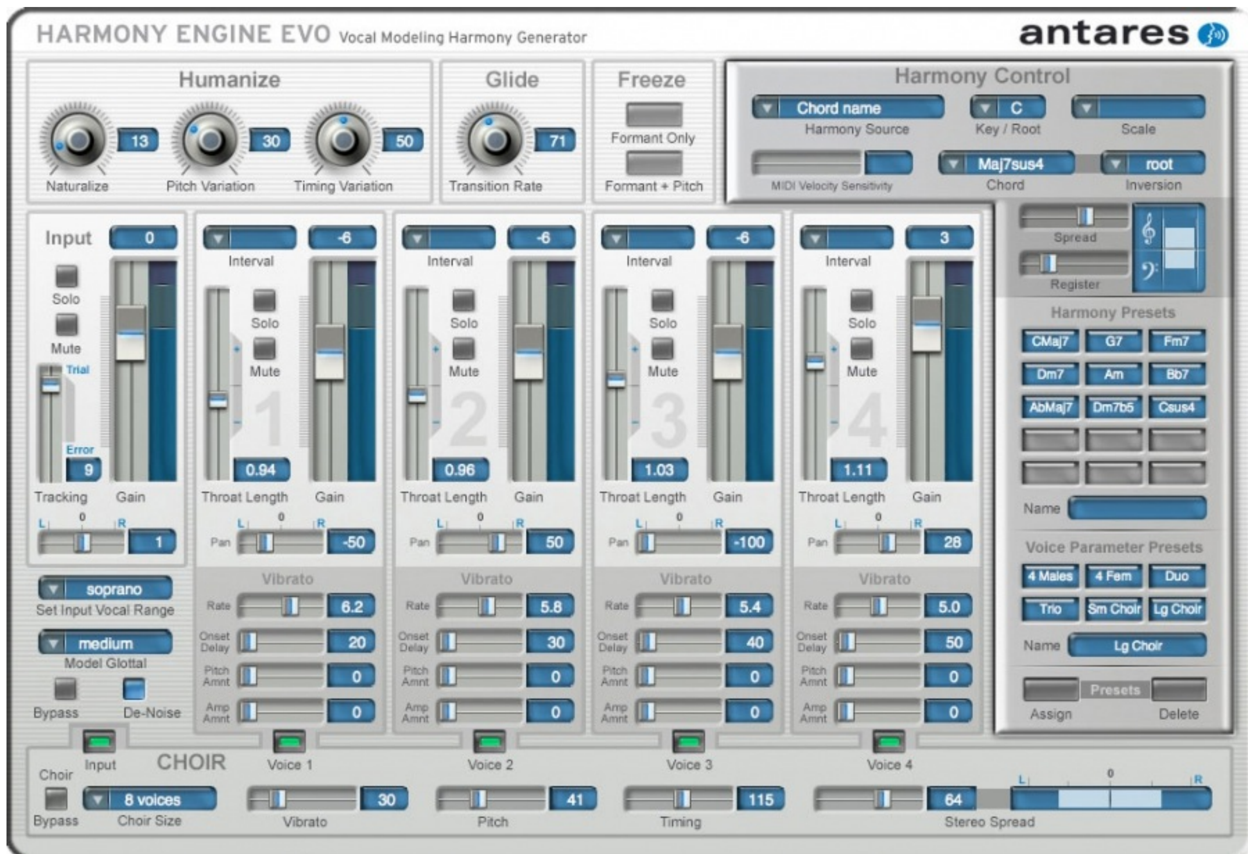
Copyright © 2015 Will Pirkle

Custom GUI Design using VSTGUI4

Will Pirkle

In this final module, we will create a VSTGUI4 GUI programmatically and without the RackAFX GUI Designer. First, have a look at a few commercially available products that implement Custom VSTGUI4 GUIs — this gives you an idea of the professionalism you can achieve with what I am calling a “pure custom” GUI. As with the rest of the RackAFX GUI Designer, the quality of your GUI is dictated mainly from the quality of your graphics files.

Harmony Engine Evo by Antares:
http://www.antarestech.com/products/detail.php?product=Harmony_Engine_Evo_4



DSP Trigger from Audio Front: http://www.audiofront.net/dspTrigger.php

phosphor from audiodamage: http://www.audiodamage.com/instruments/product.php?pid=AD027

Copyright © 2015 Will Pirkle

Rhino from Big Tick Audio Software: http://www.bigtickaudio.com/rhino/home

All of these GUIs use ordinary, subclassed, and custom view objects. This is not for the feint of heart - to achieve results like these will take you some time and you will need to dig deep into the VSTGUI4 documentation and samples you can find at the VSTGUI website as well as the mailing list. If you are serious, I would urge you to join the VSTGUI mailing list, but remember that this list is full of professionals who are using VSTGUI in their projects - please only ask questions if you have exhausted all other possibilities as the members are quite busy.

Spoiler Alert!
The GUI we are going to design will be ultra-simple, just to get you started with VSTGUI4. It will not look like the beautiful GUIs here. In order to achieve results like the ones above you are going to need to really spend some time with the library and with PhotoShop and/or KnobMan. You may want to hire a Graphic Designer to render the fundamental graphic components for you as truly professional looking GUIs usually require professionals in their design.

Graphics for your Custom GUI
You will certainly need graphics files to create nice looking GUIs. The graphics files are called “bitmaps” in VSTGUI parlance, **however they must be .PNG files for use in your GUIs**. Fortunately, converting graphic formats is commonplace today.

Copyright © 2015 Will Pirkle

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:

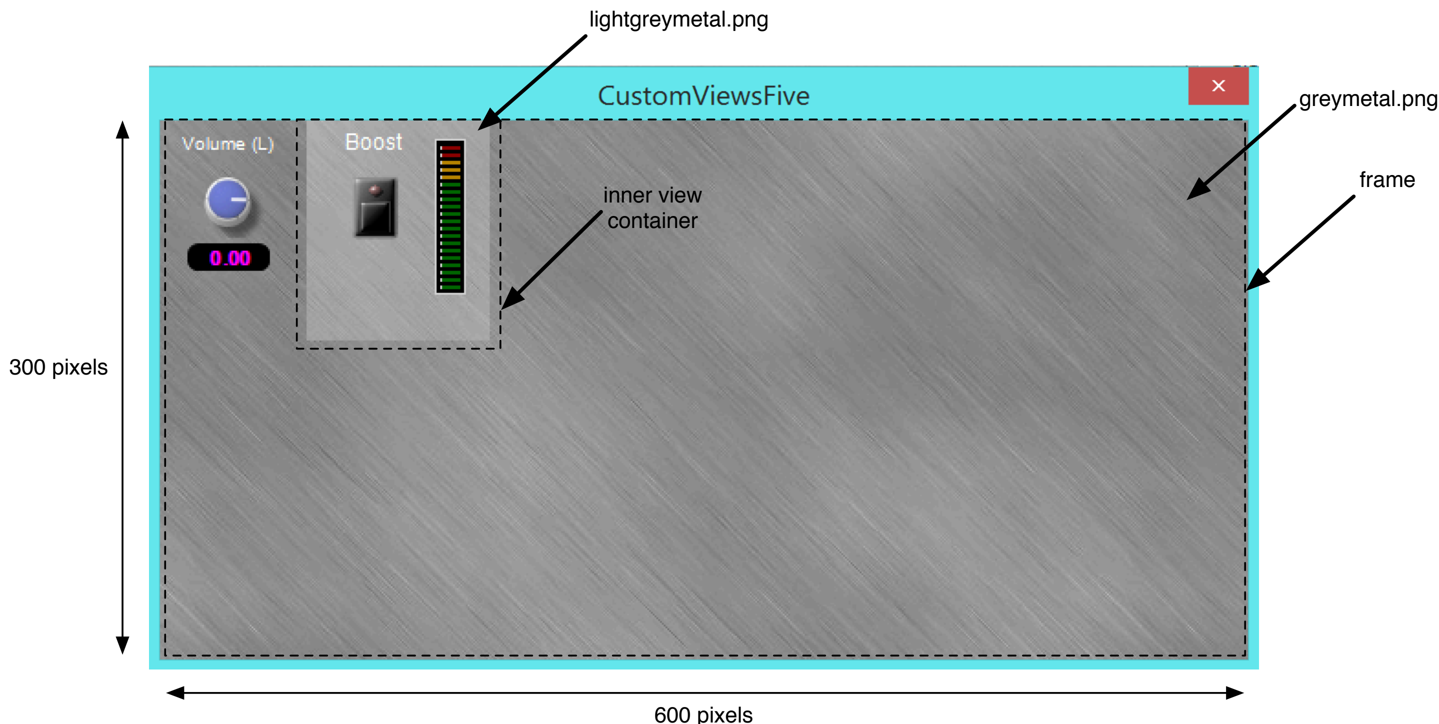
1. copy the .png file into the <project>/resources folder
2. in the Visual Studio Solution Explorer, find the <project>.rc file and right click on it; choose "View Code" and add a line to your .rc file like this (notice that the all CAPS name is identical to the lower case version):

```
/////////////////////////////////////////////////////////////////////////////
//
// 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

Anatomy of a VSTGUI GUI

VSTGUI uses some conventions regarding the setup and use of the GUI. These details are hidden from you if you use the RackAFX GUI Designer and Make VST or Make AU. However, if you are going to code your own GUI you will need to know and understand them. Here is the GUI from the accompanying project called **CustomViewsFive**:



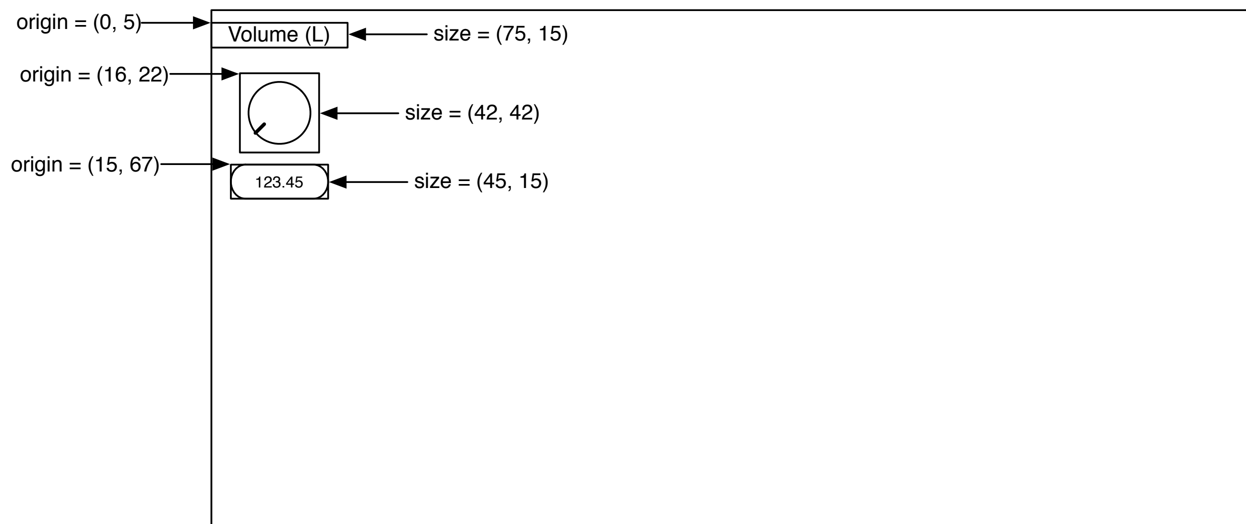
Copyright © 2015 Will Pirkle

View Containers:
There are two view containers that are shown with dotted lines around them. The outer most container has dimensions 300 x 600 pixels and has a special name: it is the ***frame*** and there is a special variable dedicated specifically for it. The *frame* view container is just like any other view container in that it holds other sub-views. We call it the *parent* of all the other views. The *frame* has no parent as it is the top level container. When we update the frame (invalidate it and make it redraw itself) it will redraw all of its sub-views. If any of the sub-views are view containers, they in turn will update their sub-views and so on. The *frame* here has a background graphic that is in the greymetal.png file. In this document, the word *frame* in italics means "the frame object" rather than frame that surrounds the text-edit and option menu controls.

There is an inner container that has the boost label, button and LED meter inside it. It has a background graphic from the lightgreymetal.png file. These PNG files are included in your plug-in automatically, but of course you can add your own graphics (see my website or YouTube videos for instructions).

Other Controls:
The other controls consist of a Left Volume label, knob and edit control, to the left of the inner view container which has its own controls embedded inside of it.

Laying out the GUI graphically:
When you design the GUI programmatically you need to do some work up-front deciding on the exact locations of the controls. When you create them, you will supply the *CRect* object that dictates their placement (top, left) and size (width, height). Without the drag-and-drop environment, this usually means getting a piece of graph paper and drawing the GUI with the origin and size values for each control. Let's look at the placement of the volume controls on the left, then we can examine the inner view container.

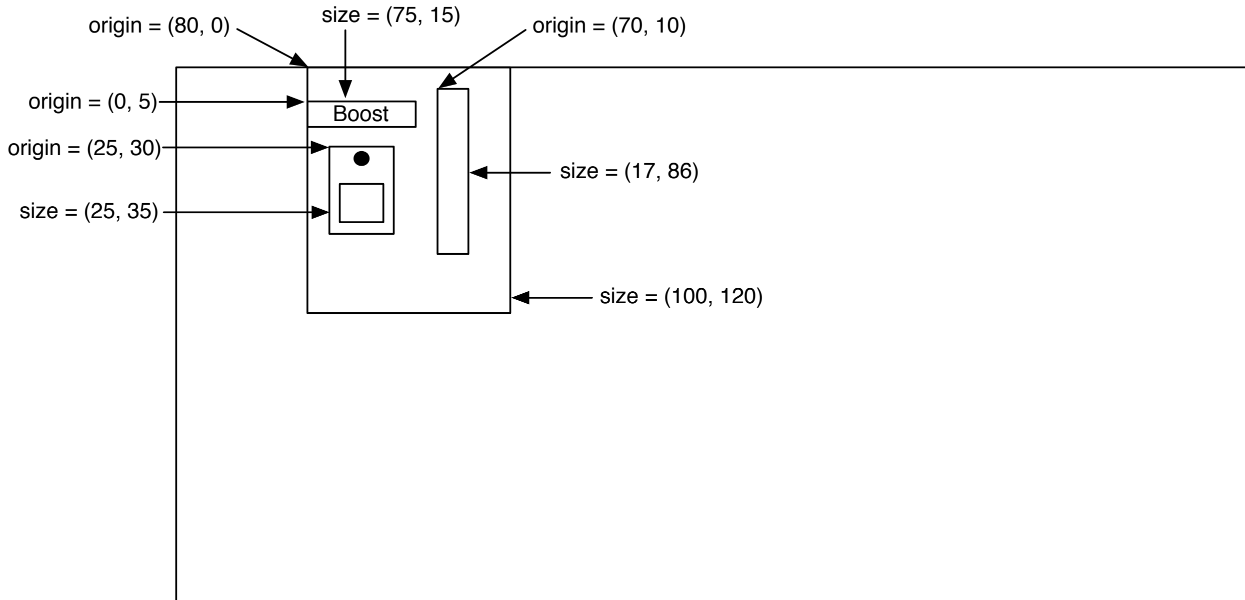


Notice that the coordinates of the origin values are relative to the *frame*, whose upper left corner is (0, 0). This is the golden rule of VSTGUI object placement — the origin of the object is always relative to the origin of the parent container. So, we can now list our first three GUI objects that will need to be created and their origin/size values:

Copyright © 2015 Will Pirkle

Object	origin	size	graphic
***frame* (CViewContainer)**	(0, 0)	(300, 600)	greymetal.png
CTextLabel	(0, 5)	(75, 15)	n/a
CAnimKnob	(16, 22)	(42, 42)	sslblue.png
CTextEdit	(15, 67)	(45, 15)	n/a

Now let's look at the inner view container and its sub-views:



The inner view container's origin is relative to the frame's origin, however, the inner view container's sub-views are positioned relative to it's origin — for example the Boost label is (0, 5) where x = 0 is the left side of the inner view container. Now we can list the rest of the controls.

Object	origin	size	graphic
CViewContainer	(80, 0)	(100, 120)	lightgreymetal.png
CTextLabel	(0, 5)	(75, 15)	n/a
COnOffButton	(25, 30)	(25, 35)	n/a
CVuMeter	(70, 10)	(17, 86)	n/a

Copyright © 2015 Will Pirkle

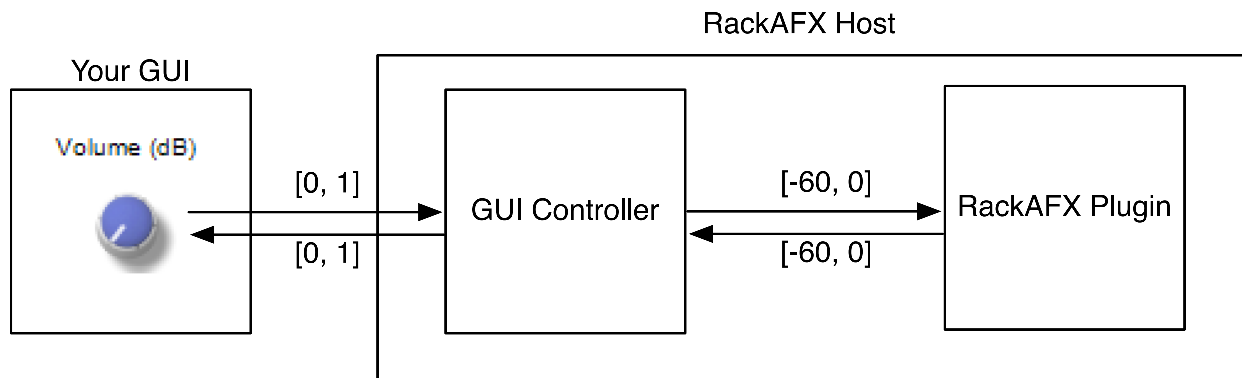
Making these tables is important — it will make the coding much faster to have these values at hand. When we create the controls, we are going to go in this order:

- create the *frame* object
  - create the *CTextLabel* object and add to the frame as a sub-view
  - create the *CAnimKnob* object and add to the frame as a sub-view
  - create the *CTextEdit* object and add to the frame as a sub-view
  - create the *CViewContainer* object and add to the frame as a sub-view
    - create the *CTextLabel* (Boost) and add it to the inner view container as a sub-view
    - create the *COnOffButton* and add it to the inner view container as a sub-view
    - create the *CVuMeter* and add it to the inner view container as a sub-view

The nesting of the bullets above mimics the way the code will flow.

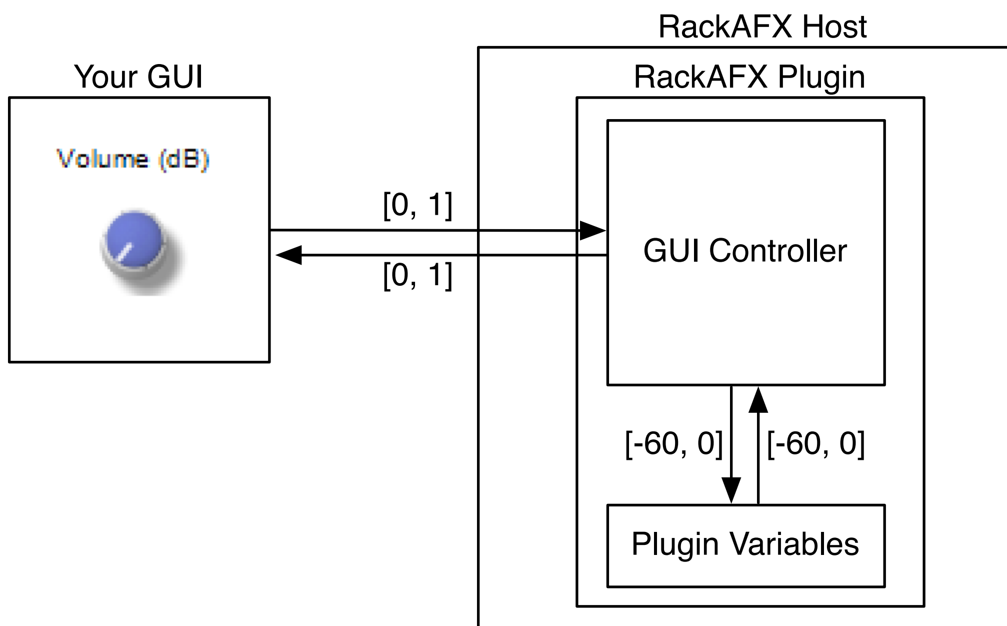
The GUI Editor Object
A special C++ object maintains the GUI — it is responsible for creating, destroying and updating the GUI controls. It is also responsible for implementing a timer that will regularly ping the GUI to update it. This not only makes the controls appear to move when you move them, it also aids in animations/movies and preset initializations. In VSTGUI, this object is called the “Editor” but I am going to refer to it as the **GUI Controller**. I am only changing the designation because the term “editor” often implies text editing. The GUI Controller object translates VSTGUI values and applies them to the plug-in variables. It also animates the meter object.

You may remember this figure from module 3:



In this case, the GUI Controller object is part of the RackAFX host. For your custom GUI, the location of the object changes — your plugin now owns the GUI Controller object:

Copyright © 2015 Will Pirkle



Before we look at the GUI Controller object, we need to talk about how the variables are being stored on your RackAFX plugin. This information is in the Appendix A of my FX book. I have repeated part of that here.

"When you set up a control in RackAFX, you are really creating a new C++ object that is added to a linked list. The linked list is a member of your CPlugIn class named **m_UIControlList**. It contains a list of C++ **CUICtrl** objects. A CUICtrl object can represent one of the following:

- Slider
- Radio button bank
- Meter

The simplest way to explain is by example. Look in your initUI() method for any Plug-In and you will see the instantiation and initialization of the GUI objects. Here's the first part of the Volume slider code from the very first Project. The highlights are in bold. Remember, <u>do not ever edit this code manually</u>.

```
m_fVolume = 0.750000;
CUICtrl ui0;
ui0.uControlType = FILTER_CONTROL_CONTINUOUSLY_VARIABLE;
ui0.uControlId = 0;
ui0.fUserDisplayDataLoLimit = 0.000000;
ui0.fUserDisplayDataHiLimit = 1.000000;
ui0.uUserDataType = floatData;
ui0.fInitUserIntValue = 0;
ui0.fInitUserFloatValue = 0.750000;
ui0.fInitUserDoubleValue = 0;
ui0.fInitUserUINTValue = 0;
ui0.m_pUserCookedIntData = NULL;
ui0.m_pUserCookedFloatData = &m_fVolume;
ui0.m_pUserCookedDoubleData = NULL;
```

Copyright © 2015 Will Pirkle

```
ui0.m_pUserCookedUINTData = NULL;
ui0.cControlUnits = "                                   ";
ui0.cVariableName = "m_fVolume";
ui0.cEnumeratedList = "SEL1,SEL2,SEL3";
ui0.dPresetData[0] = 0.000000;ui0.dPresetData[1] <SNIP SNIP SNIP>
ui0.cControlName = "Volume";
```

The very first line initializes the underlying variable, m_fVolume as you set it up when you created the Slider. … The uControlID is 0 – this is the value that is passed to userInterfaceChange() when a control is manipulated. This "unique ID" paradigm for identifying the control is universal among the other Plug-In APIs. The connection to the variable itself is via a pointer. Since this is a float variable, the *m_pUserCookedFloatData* is set to the address of the underlying variable. This is how RackAFX manipulates it for you."

You are going to need to access this list of objects to get and set the underlying values of variables as well as access VU meter variables. I have added some helper functions to make this as simple as possible.

Getting a pointer to one of these objects is easy - you just need to know the Control ID value used in *userInterfaceChange()*. Here is an example of getting the *CUICtrl** for the first LED Meter which has the Control ID of 50:

```
CUICtrl* pUICtrl = m_pPlugIn->getUICtrlByControlID(50);
```

```
float getNormalizedValue(CUICtrl* pUICtrl)
```
This method returns the normalized value of the variable connected to this *CUICtrl* object.

```
void setPlugInParameterNormalized(CUICtrl* pUICtrl, float value)
```
This method converts the normalized *value* parameter for you, sets it in the plug-in's *CUICtrl* object, and then calls *userInterfaceChange()* after the update is done.

We will look at examples of using these functions a bit later. But, they do most of the low level work for you.

There are actually a few ways to deal with the GUI Controller object, but the simplest way that is platform independent is to derive your GUI Controller object from the following VSTGUI4 objects:

VSTGUIEditorInterface
This object is the owner of the *frame* object. It only has a few variables and methods. The most important are shown in **bold**.

- *getKnobMode()* — this returns a constant that tells how the knob is handled when you move the mouse. The default is the linear mode where linear movement over the knob causes it to rotate. The full set of choices is:
  kLinearMode;
  kRelativCircularMode;
  kCircularMode;
- *getFrame()* — this returns a pointer to the frame object

CFrame frame; — *CFrame* is just a special kind of *CViewContainer* that is designed to be the outermost container.

Copyright © 2015 Will Pirkle

```
class VSTGUIEditorInterface
{
public:
      virtual void doIdleStuff () {}
      virtual int32_t getKnobMode () const { return 0; }

      virtual void beginEdit (int32_t index) {}
      virtual void endEdit (int32_t index) {}

      ///< frame will change size, if this returns false the upstream imple
      ///  mentation does not allow it and thus the size of the frame will
      ///  not change
      virtual bool beforeSizeChange (const CRect& newSize,
                                     const CRect& oldSize) { return true; }


      virtual CFrame* getFrame () const { return frame; }
protected:
      VSTGUIEditorInterface () : frame (0) {}
      virtual ~VSTGUIEditorInterface () {}

      CFrame* frame;
};
```

CControlListener
The GUI objects will deliver control change information to their *CControlListener* buddies. In the previous modules, the listener was the RackAFX GUI (or VST3/AU wrapper objects). Now, your object must handle these chores. *CControlListener* is pure abstract, and we need to override the single pure abstract method *valueChanged()* shown in bold, which is the message handler that is called when a control changes due to user interaction.

```
class CControlListener
{
public:
      virtual ~CControlListener() {}
      virtual void valueChanged (VSTGUI::CControl* pControl) = 0;
      virtual int32_t controlModifierClicked (VSTGUI::CControl* pControl,
                                    VSTGUI::CButtonState button)
                                    { return 0; }

      virtual void controlBeginEdit (VSTGUI::CControl* pControl) {}
      virtual void controlEndEdit (VSTGUI::CControl* pControl) {}
      virtual void controlTagWillChange (VSTGUI::CControl* pControl) {}
      virtual void controlTagDidChange (VSTGUI::CControl* pControl) {}
#if DEBUG
      virtual char controlModifierClicked (VSTGUI::CControl* pControl, long
button) { return 0; }
#endif
};
```

The *valueChanged()* method might be the most important function we need to write. It needs to do several things each time a new control change message is received:

- decode the control-tag to figure out when control changed

Copyright © 2015 Will Pirkle

- find the plug-in variable that is connected to this control-tag
- translate the normalized value from the control to the required value for the plug-in
- change the plug-in's underlying variable accordingly
- call *userInterfaceChange()* to alert the plug-in
- broadcast this control change to all GUI objects that have the same control-tag

The last item is crucial — it is the glue that binds the GUI controls together that share a common control-tag. This is why the text edit control updates when you move the associated knob and it is up to you to make that connection happen in code.

CBaseObject

This is the base object for all VSTGUI4 views, controls and control listeners. It takes care of reference counting and you don't have to worry about any of it.

CVSTGUIController

CVSTGUIController is the object that I have created for you to use as your GUI Controller. When you open an old project or create a new project, the CVSTGUIController.h and CVSTGUIController.cpp files will be added to your project folder, but NOT to your Visual Studio project — you must manually add these files the same way you added the other advanced GUI API files. I had several things in mind when creating this object for you:

- it needs to be platform-independent and able to easily integrate with Audio Unit plugins, which have a bit more complex connections (you don't have to worry about any of it, but the AU-specific code is there if you want to see it)
- it features three important functions that are critical to the object lifecycle:
  - *open()* — called to create the *frame* and populate it with controls
  - *close()* — called before the *frame* is destroyed for cleanup
  - *idle()* — update the *frame* in response to the GUI timer ping
- it needs to implement the overrides from the two main base class objects:
  - *getKnobMode()* from *VSTGUIEditorInterface*
  - *valueChanged()* from *CControlListener*

In addition, I have also included a bunch of useful functions that handle much of the low-level chores of translating normalized to plain values, working with bitmaps, and handling non-linear controls (volt/octave and log-based).

There are several important member variables too:

```
protected:
	void* m_hPlugInInstance;// HINSTANCE of this DLL (WinOS only)

	// --- our plugin
	CPlugIn* m_pPlugIn;

	// --- timer for GUI updates
	CVSTGUITimer* timer;
```

void* m_hPlugInInstance

The first variable is the instance handle for the DLL (Windows) OR a pointer to the Audio Unit instance (AU only). This is passed into the *open()* function and is required to create the *frame* object in Windows. The instance handle is delivered to the plugin when it is loaded. You don't have to worry about its details but you do have to store it.

Copyright © 2015 Will Pirkle

CPlugIn* m_pPlugIn
This is a pointer to the plug-in and is the mechanism that we use to connect the GUI to the plug-in. This is one of several different ways of making the connection. We pass our ***this*** pointer to the GUI Controller when we create the GUI. The GUI Controller then uses it to get and set variables on our object as well as call our *userInterfaceChange()* method. Some may criticize this method of binding the objects together, but it is the simplest and most straightforward. After you have the GUI Controller object working with the plug-in, you might think of other ways to make the connection happen.

CVSTGUITimer* timer
We need a platform-independent timer object to update the GUI on an interval that I have chosen to be 50 milliseconds, though you may change that. The normal VSTGUI update interval is 300 milliseconds, but many feel that this is too slow and causes the GUI and its animations to be slightly glitchy. You can experiment with the interval — long intervals will use less processing time, but look less-smooth.

You have two options when dealing with my built-in GUI Controller object - modify it directly or subclass it. Since you always get a fresh, blank object with each plug-in project, modifying it directly is the easiest and that is what we do here. Here is the declaration of the *CVSTGUIController* object without the miscellaneous helper functions:

```
#include "../vstgui4/vstgui/vstgui.h"
#include "plugin.h"
#include <cstdio>
#include <string>
#include <vector>
#include <map>

#ifdef AUPLUGIN
    #include <AudioToolbox/AudioToolbox.h>
#endif

using namespace std;

namespace VSTGUI {

class CVSTGUIController : public VSTGUIEditorInterface,
                          public CControlListener,
                          public CBaseObject
{
public:
      CVSTGUIController();
      virtual ~CVSTGUIController();

#ifdef AUPLUGIN
      // --- the AU for preset change notification
    AudioUnit m_AUInstance;
    AUEventListenerRef m_AUEventListener;
#endif

      // --- timer notification callback
      CMessageResult notify(CBaseObject* sender, const char* message);

      // --- open function:
      bool open(void* window, CPlugIn* pPlugIn, int& nWidth, int& nHeight,
                void* hPlugInInstance = NULL);
```

Copyright © 2015 Will Pirkle

```
	// --- close function; destroy frame and forget timer
	void close();

	// --- do idle processing
	void idle();

	// --- function to create/initialize/destroy the controls
	void createControls();
	void initControls(bool bSetListener = false);// bSetListener is AU only

	// --- VSTGUIEditorInterface override
	virtual int32_t getKnobMode() const;

	// --- CControlListener override (pure abstract, so must)
	virtual void valueChanged(VSTGUI::CControl* pControl);

	// --- get a bitmap
	CBitmap* getBitmap(const CResourceDescription& desc, CCoord left = -1,
	                   CCoord top = -1, CCoord right = -1,
	                   CCoord bottom = -1);

	// --- plugin helpers
	float getNormalizedValue(CUICtrl* pUICtrl);

	// --- GUI SPECIFIC CONTROL POINTERS: ADD YOURS HERE




	// --- END GUI SPECIFIC CONTROL POINTERS

protected:
	void* m_hPlugInInstance; // HINSTANCE of this DLL (WinOS only)

	// --- our plugin
	CPlugIn* m_pPlugIn;

	// --- timer for GUI updates
	CVSTGUITimer* timer;

// --- miscellaneous functions
public:
	etc…
```

Plugin-Side Code

Before we get to the details of implementing the GUI Controller object, let’s look at the plug-in side portion of the code since it is easier to understand. In the plugin’s .h file, we declare a pointer to our *CVST-GUIController* object:

```
// un-comment for advanced GUI API: see www.willpirkle.com for details and
// sample code
#include "GUIViewAttributes.h"
#include "../vstgui4/vstgui/vstgui.h"

// un-comment for pure custom VSTGUI: see www.willpirkle.com for details and
// sample code
#include "VSTGUIController.h"
```

Copyright © 2015 Will Pirkle

```
class CCustomViewsFive : public CPlugIn
{
public:
	// RackAFX Plug-In API Member Methods:

	<SNIP SNIP SNIP>

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

	// Add your code here: -------------------------------------------- //
	CVSTGUIController* m_pGUIController; // full custom VSTGUI4 GUI

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

	etc…
```

In the plugin's .cpp file, examine the following:

Constructor:

- NULL the pointer

```
CCustomViewsFive::CCustomViewsFive()
{
	// 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
	m_pGUIController = NULL;

}
```

showGUI():

- respond to the GUI_HAS_USER_CUSTOM message by setting the *info->bHasUserCustomView* flag to true
- respond to the GUI_USER_CUSTOM_OPEN message by instantiating the *CVSTGUIController* member object and calling its *open()* method (we will discuss the *open()* arguments shortly)
- respond to the GUI_USER_CUSTOM_CLOSE message by closing the GUI and deleting the GUI Controller

NOTE: for pure custom GUI's, the messages GUI_DID_OPEN, GUI_WILL_CLOSE, and GUI_TIMER_PING are not called.

```
void* __stdcall CCustomViewsFive::showGUI(void* pInfo)
{
	// --- ALWAYS try base class first in case of future updates
```

Copyright © 2015 Will Pirkle

```
	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_HAS_USER_CUSTOM:
		{
			// --- set this variable to true if you have a custom GUI
			info->bHasUserCustomView = true; // yes, we have one!
			return NULL;
		}

		//	open() sets the new size of the window in info->size
		//	return a pointer to the newly created object
		case GUI_USER_CUSTOM_OPEN:
		{
			m_pGUIController = new CVSTGUIController;
			if(m_pGUIController)
			{
				m_pGUIController->open(info->window,
							this,
							info->size.width,
							info->size.height,
							info->hPlugInInstance);
			}
			return m_pGUIController;
		}

		// --- call the close() function and delete the controller object
		case GUI_USER_CUSTOM_CLOSE:
		{
			if(m_pGUIController)
			{
				m_pGUIController->close();
				delete m_pGUIController;
				m_pGUIController = NULL;
			}
			return m_pGUIController; // returning NULL = success
		}

		// --- handle paint-specific timer stuff
		case GUI_TIMER_PING:
		{

			return NULL;
		}
	}

	return NULL;
}
```

Copyright © 2015 Will Pirkle

That's all there is for the plug-in side code. Now it's time to dig into the GUI Controller object!

GUI Controller-Side Code

The first thing you need to remember is that you will be instantiating all the GUI objects, and because you need to implement the *valueChanged()* method, you must cache the pointers to those objects. The *valueChanged()* method's only argument is a pointer to the control that changed, so you can match pointers to figure out what changed. In a larger GUI, I would use an array, vector or other list-type object to hold the pointers. However, since we only have a few controls, I have declared them individually. In the GUI Controller's .h file, you can find the declarations:

```
CTextLabel*       m_pVolLeftLabel;
CAnimKnob*        m_pVolLeftKnob;
CTextEdit*        m_pVolLeftEdit;

CViewContainer*   m_pBoostVC;
CTextLabel*       m_pBoostLabel;
COnOffButton*     m_pBoostButton;
CVuMeter*         m_pLeftMeter;
```

You will notice I have declared two member functions *createControls()* and *initControls()* to break the code into smaller chunks. In addition the *initControls()* method will be called when the user selects a preset.

The timer notification callback function (specified by the VSTGUI timer object) is also declared:

```
CMessageResult notify(CBaseObject* sender, const char* message);
```

VSTGUIController.cpp File

The rest of the implementation is in the object's .cpp file. The best way to learn this is probably by example, so let's step through the object one function at a time.

Constructor:

- NULL the object pointers so we don't accidentally use them
- create our timer object

```
CVSTGUIController::CVSTGUIController()
{
      m_pVolLeftLabel = NULL;
      m_pVolLeftKnob = NULL;
      m_pVolLeftEdit = NULL;

      m_pBoostVC = NULL;
      m_pBoostLabel = NULL;
      m_pBoostButton = NULL;
      m_pLeftMeter = NULL;

      // create a timer used for idle update
      timer = new CVSTGUITimer(dynamic_cast<CBaseObject*>(this));
}
```

Destructor:

- call *forget()* on the timer so reference counting will release it — do NOT delete the object

Copyright © 2015 Will Pirkle

```
CVSTGUIController::~CVSTGUIController()
{
      // --- stop timer
      if(timer)
            timer->forget();
}
```

notify()

This method is the timer callback function. All we need to do is call the *idle()* function on our *frame* object, which will in turn call the redrawing methods on the sub-views as needed:

```
CMessageResult CVSTGUIController::notify(CBaseObject* /*sender*/,
                                         const char* message)
{
    if(message == CVSTGUITimer::kMsgTimer)
    {
        if(frame)
            idle();

        return kMessageNotified;
    }
    return kMessageUnknown;
}
```

getKnobMode()

This little function just returns the knob mode constant - here use use the linear knob control.

```
int32_t CVSTGUIController::getKnobMode() const
{
      /* choices are:   kLinearMode;
                        kRelativCircularMode;
                        kCircularMode; */

      return kLinearMode;
}
```

open()

This is the function that creates the *frame* object and all the sub-views. We will use the *createControls()* function to create all the interior sub-views. The *open()* method prototype is:

```
bool open(void*   window,
          CPlugIn* pPlugIn,
          int&     nWidth,
          int&     nHeight,
          void*    hPlugInInstance)
```

`void* window`

This is a pointer to the parent window of the GUI. It is passed into your plug-in in the VSTGUI_VIEW_INFO struct's *window* parameter. For WinOS, this is a *HWND** and for MacOS, it is a *NSView**. If you don't know what those are, don't worry as you won't need to deal with them after this function is called.

`CPlugIn* pPlugIn`

This is our plug-in object; we need to store it to use in almost all of the other methods.

Copyright © 2015 Will Pirkle

`int&  nWidth, nHeight`
These two references are output variables which tell the host how to size the window according to our internal dimensions. You must remember to set these variables so the host can properly size and position our GUI window.

`void* hPlugInInstance`
We already discussed it - for WinOS you need to store this on an externally declared variable.

The **open()** function needs to do the following:
- store the pointer to the RackAFX plugin
- set the plug-in instance variable (WinOS only)
- set the dimensions on the return arguments
- create the *frame* object
- set the *frame's* background color and/or bitmap graphic
- populate the *frame* with GUI controls
- initialize the controls
- set the timer interval and start it

Here are some code chunks that perform these activities:
- store the pointer to the RackAFX plugin
- set the plug-in instance variable
- set the dimensions on the return arguments

```
bool CVSTGUIController::open(void* window, CPlugIn* pPlugIn, int& nWidth,
                             int& nHeight, void* hPlugInInstance)
{
      if(!window) return false;

      m_pPlugIn = pPlugIn;

#if MAC && AUPLUGIN
      m_AUInstance = (AudioUnit)hPlugInInstance;
#else
      m_hPlugInInstance = hPlugInInstance;
#endif

      // --- set the return variables (you may want to store them too)
      nWidth = 600;
      nHeight = 300;
```

- create the *frame* object
- set the *frame's* background color

```
      //-- create the frame rect: it dictates the size in pixels
      CRect frameSize(0, 0, nWidth, nHeight);

      // --- construct the frame
      frame = new CFrame(frameSize, this);

      // --- open it
#if defined _WINDOWS || defined WINDOWS || defined _WINDLL
    frame->open(window, kHWND);      // for WinOS, window = HWND
#else
    frame->open(window, kNSView);    // for MacOS, window = NSView*
```

Copyright © 2015 Will Pirkle

```
#endif

      // --- set the frame background color and/or image

      // --- example with built-in color
      frame->setBackgroundColor(kWhiteCColor);

      // --- example with r,g,b,a
      //     here it is red with semi-transparency,
      //     you will see the black (not white) background behind it
      frame->setBackgroundColor(CColor(255, 0, 0, 128));
```

- create the background graphic; here I am using a tiled graphic that requires the *CNinePartTiledBitmap* object, but I show commented code for regular bitmaps

```
/* UNCOMMENT THIS TO SEE THE NORMAL BITMAP, AND COMMENT THE CHUNK BELOW!
      CBitmap* pBitmap = getBitmap("greymetal.png");

      // --- always check pointer!
      if(pBitmap)
      {
            // --- set it
            frame->setBackground(pBitmap);

            // --- and... forget it (VSTGUI uses reference counting)
            pBitmap->forget();
      }
*/

      // --- now do a tiled version, all coords = 0 gives ordinary, infinite
      //     tiling in each dimension
      //     I recommend not doing exotic tiling because the rendering is
      //     very slow
      //
      // --- example of tiled bitmap
      CBitmap* pTiledBitmap = getBitmap("greymetal.png", 0, 0, 0, 0);

      // --- always check pointer!
      if(pTiledBitmap)
      {
            // --- set it
            frame->setBackground(pTiledBitmap);

            // --- forget: VSTGUI uses reference counting
            pTiledBitmap->forget();
      }
```

- populate the *frame* with GUI controls (we'll look at the method shortly)
- initialize the controls
- set the timer interval and start it

```
      // --- now that the frame has a background, continue with controls
      //     I made this a separate function because it is usually very long
      createControls();
```

Copyright © 2015 Will Pirkle

```
	// --- the main control inits
	initControls(true); // true = setup AU Listener (only once)

	// --- set/start the timer
	if(timer)
	{
	  timer->setFireTime(METER_UPDATE_INTERVAL_MSEC);
	  timer->start();
	}

	return true;
}
```

close()

This method is refreshingly short. We need to:

- stop the timer
- forget the frame, after setting it to NULL

```
void CVSTGUIController::close()
{
	if(!frame) return;

	// --- stop timer
	if(timer)
		timer->stop();

	//-- on close we need to delete the frame object.
	//-- once again we make sure that the member frame variable is set to
	//   zero before we delete it
	//-- so that calls to setParameter won't crash.
	CFrame* oldFrame = frame;
	frame = 0;
	oldFrame->forget(); // this will remove/destroy controls
}
```

idle()

This is actually one of the most important functions we need to implement. First, it is responsible for re-painting the GUI each time the timer fires (and assuming there is something to repaint — if no controls have changed, they won't repaint themselves which prevents the screen from flickering). Second, here is where we handle the RackAFX function called *sendUpdateGUI().* This function allows you to alter the underlying RackAFX variable, then call the function which updates the GUI based on the RackAFX variable. This function works in all versions — RackAFX, VST and AU.

Re-paint update:

Here we need to do any processing that requires repainting or updating of our view based controls like VU Meters or audio waveform/spectrum graphs. We have one LED meter to deal with, so we need to update that object with the current value of it's *m_pCurrentMeterValue* variable. After we take care of our objects, we need to call the *idle()* method on the *frame* so it can update its views.

***sendUpdateGUI()*:**

To deal with this message we need to check the somewhat hidden send-update-GUI-flag. This flag is located in a special array in RackAFX named *m_uPluginEx*. The index of the variable is defined in your pluginconstants.h file as UPDATE_GUI. You check the flag — if it is set, just call *initControls()* to update

Copyright © 2015 Will Pirkle

the GUI all at once using the current values of the RackAFX variables. You don't have to worry about flickering as controls that do not need redrawing will not be repainted.

```
void CVSTGUIController::idle()
{
	if(!m_pPlugIn) return;

	// --- VU Meter objects need to have their value set here so they ani
	//     mate properly
	//     Our VU Meter is connected to the RAFX Control ID 50
	//
	CUICtrl* pUICtrl = m_pPlugIn->getUICtrlByControlID(50);
	if(pUICtrl)
	{
		// --- the meter value is in pUICtrl->m_pCurrentMeterValue
		if(m_pLeftMeter)
			m_pLeftMeter->setValue(*pUICtrl->m_pCurrentMeterValue);
	}

	// --- handle sendUpdateGUI() from the plug-in
	//     check the UPDATE_GUI flag
	if(m_pPlugIn->m_uPlugInEx[UPDATE_GUI] == 1)
	{
		// --- update the controls
		initControls();

		// --- reset flag
		m_pPlugIn->m_uPlugInEx[UPDATE_GUI] = 0;
	}

	// --- then, update frame -
	if(frame)
		frame->idle();
}
```

There is a button on the RackAFX interface labeled **Update GUI** that you can use to test the *sendUpdateGUI()* functionality. It sets the Volume Left to -10.0db and Boost to ON when depressed, then resets the Volume Left to 0.0dB and Boost to OFF when pressed again.

createControls()

In this function, we will create all the GUI objects. You need to have gone through modules 3-6 to understand how these constructors work — they are repeats of the lessons you learned in those modules so I will not go over every detail; here it is important to focus on how the views are added to the *frame* or inner view container objects. So, lets step through the function a piece at a time. In the first part, we just check to make sure we have a valid plug-in buddy to connect to, and call the frame's *onActivate()* method if there is no plug-in. We will call the *onActivate()* method at the end of initControls() after adding all the views.

```
void CVSTGUIController::createControls()
{
	if(!frame)
		return;

	if(!m_pPlugIn)
		return frame->onActivate(true);
```

Copyright © 2015 Will Pirkle

Now we can begin instantiating the control objects. Start with the Volume Left text label; create it and add it to the *frame* object — note how the rectangle is set using the origin and size parameters we wrote in our tables previously. This example also has some commented code to let you play with setting styles and text after creation:

```
// --- add the text label at origin(0,5) size(75,15)
/*
      Constructor:
      CTextLabel(const CRect& size,
                 UTF8StringPtr txt = 0,
                 CBitmap* background = 0,
                 const int32_t style = 0)

As an example, I'll create it first, then set the text rather than us
ing text in constructor*/

CPoint labelOrigin(0,5);
CPoint labelSize(75,15);
CRect labelRect(labelOrigin, labelSize);

// --- create
m_pVolLeftLabel = new CTextLabel(labelRect);

// --- set extra attributes; see CTextLabel & CParamDisplay; there are
//     MANY attributes you can set on CParamDisplay
//     objects!
if(m_pVolLeftLabel)
{
      // --- set font color
      m_pVolLeftLabel->setFontColor(kWhiteCColor);

      // --- set background transparent
      m_pVolLeftLabel->setTransparency(true);

      // --- set the text
      m_pVolLeftLabel->setText("Volume (L)");

      // --- OPTIONAL set the font - if you don't set it, you get plat
      //     form default (Ariel 10)
      //
      // In this example, I will use a default font
      m_pVolLeftLabel->setFont(kNormalFontSmaller);

      // --- OPTIONAL set the style
      // pLabel->setStyle(pLabel->getStyle() | k3DOut);

      // --- do any more customization, then add to frame
      frame->addView(m_pVolLeftLabel);
}
```

We continue with the knob object. Notice how the control-tag is set manually — it is the Control ID in RackAFX, so it matches the values in *userInterfaceChange().* **Notice the use of the helper function *getBitmap()* to instantiate the bitmap object:**

```
CPoint knobOrigin(16, 22);
CPoint knobSize(42, 42);
CRect knobRect(knobOrigin, knobSize);
```

Copyright © 2015 Will Pirkle

```
int nTag = 8; // RAFX ControlID for Volume (L)
int nPixMaps = 80;
int nHtOneImage = 42;

// --- get the bitmap
CBitmap* pBitmap = getBitmap("sslblue.png");

// --- if the bitmap does not exist DO NOT CREATE the control!
if(pBitmap)
{
      // --- create it; leave offset at (0,0) - it shifts top,left of
      //     control if fine adjustment needed
      m_pVolLeftKnob = new CAnimKnob(knobRect, this, nTag, nPixMaps,
                                    nHtOneImage, pBitmap);

      // --- add to frame
      frame->addView(m_pVolLeftKnob);

      // --- forget, VSTGUI uses reference counting
      pBitmap->forget();
}
```

And, now the *CTextEdit* control, just under the knob — as with the custom view example, there are many attributes you can set with this control:

```
// --- add the edit control at origin(15,67) size(45, 15)
/*
      Preferred Constructor:
      CTextEdit(const CRect& size,
                  CControlListener* listener,
                  int32_t tag,
                  UTF8StringPtr txt = 0,
                  CBitmap* background = 0,
                  const int32_t style = 0);

      The styles are the same as a CTextLabel, which CTextEdit is de
      rived from. There are MANY styles that can be set, making the
      edit control rounded, with or without frame, etc...
      See docs and experiment!
*/
CPoint editOrigin(15, 67);
CPoint editSize(45, 15);
CRect editRect(editOrigin, editSize);

// --- create it: use same nTag variable to link knob/edit
//     "0.00" is initial text
m_pVolLeftEdit = new CTextEdit(editRect, this, nTag, "0.00");

// --- customize; black background and magenta font color
if(m_pVolLeftEdit)
{
      // --- this is an example of using a non built-in font
      //
      // --- first create the font description
      CFontDesc* fontDesc = new CFontDesc("Microsoft Sans Serif", 10);

      if(fontDesc)
```

Copyright © 2015 Will Pirkle

```
        {
              // --- you can change the style - here is bold
              //     be careful here - some styles don't work well in
              //     edit control depending on the font size
              //     for example if font size is too big, italic text
              //     will move around as it changes
              fontDesc->setStyle(fontDesc->getStyle() | kBoldFace);

              // --- this should be named createPlatformFont - it creates
              //     the font on different platforms
              fontDesc->getPlatformFont();

              // --- set the new font
              m_pVolLeftEdit->setFont(fontDesc);
        }

        // --- set back and font colors
        m_pVolLeftEdit->setBackColor(kBlackCColor);
        m_pVolLeftEdit->setFontColor(kMagentaCColor);

        // --- OR you can make the background transparent
        // pEdit->setTransparency(true);

        // --- give it rounded corners with round radius of 5
        //
        //     RoundRect is a style
        m_pVolLeftEdit->setStyle(m_pVolLeftEdit->getStyle() |
                                 kRoundRectStyle);

        // --- round rect radius is an attribute in pixels
        m_pVolLeftEdit->setRoundRectRadius(5);

        // --- add to the frame
        frame->addView(m_pVolLeftEdit);
  }
```

We have one more GUI object to add to the *frame*: the inner view container that holds the boost button and LED meter. Notice the origin is given in *frame* coordinates, that is the view container is positioned relative to the *frame's* origin. It's creation is just like what we saw in the custom view except here we give it a background graphic:

```
  // --- EXAMPLE OF CREATING A VIEW CONTAINER
  /*
        A powerful feature of VSTGUI4 is the ability to create a view
        container and add views to it.
        You can then move the whole container around or show/hide it and
        all subviews will also move/show/hide.
        You can have any number of nested containers.

        In this example, we'll create a view container with a switch
        (COnOffButton) and a LED VU Meter inside it.
  */

  // --- add the edit control at origin(15,67) size(45, 15)
  /*
        Preferred Constructor:
        CViewContainer(const CRect& rect)
  */
```

Copyright © 2015 Will Pirkle

```
    CPoint vcOrigin(80, 0);
    CPoint vcSize(100, 120);
    CRect vcRect(vcOrigin, vcSize);

    // --- create
    m_pBoostVC = new CViewContainer(vcRect);
    if(m_pBoostVC)
    {
        // --- set background bitmap, color, or use setTransparency for
        //     transparent VC
        CBitmap* pBitmap = getBitmap("lightgreymetal.png");

        // --- if the bitmap does not exist DO NOT CREATE the control!
        if(pBitmap)
        {
            // --- this is also how to change a bitmap on the fly
            m_pBoostVC->setBackground(pBitmap);

            // --- always forget
            pBitmap->forget();
        }
```

Notice we have not added the closing curly bracket here — we are still inside the if(m_pBoostVC) statement. It is here that we will add the inner controls. After the controls are added, we will add the populated view container to the *frame.* We will start with the Boost label — its origin is relative to the inner view container, and we add it to the container, not the *frame* object:

```
        // --- add controls: note that origin() points are relative to
        //     VC, not frame
        CPoint labelOrigin(0,5);
        CPoint labelSize(75,15);
        CRect labelRect(labelOrigin, labelSize);

        // --- create label
        m_pBoostLabel = new CTextLabel(labelRect);

        // --- set extra attributes;
        if(m_pBoostLabel)
        {
            // --- set font color
            m_pBoostLabel->setFontColor(kWhiteCColor);

            // --- set background transparent
            m_pBoostLabel->setTransparency(true);

            // --- set the text
            m_pBoostLabel->setText("Boost");

            // --- do any more customization, then add to VC, not the
            //     frame
            m_pBoostVC->addView(m_pBoostLabel);
        }
```

Now, we add the on/off button:

```
        // --- add the COnOffButton origin(25,30) size(25, 35)
        /*
```

Copyright © 2015 Will Pirkle

```
        Preferred Constructor:
        COnOffButton(const CRect& size,
                            CControlListener* listener = 0,
                            int32_t tag = -1,
                            CBitmap* background = 0,
                            int32_t style = 0);
*/
CPoint buttonOrigin(25, 30);
CPoint buttonSize(25, 35);
CRect buttonRect(buttonOrigin, buttonSize);
int nTag = 45; // RAFX ControlID for BOOST

// --- set background bitmap, color, or use setTransparency for
//     transparent VC
pBitmap = getBitmap("medprophetbutton.png");

// --- if the bitmap does not exist DO NOT CREATE the control!
if(pBitmap)
{
        // --- create button
        m_pBoostButton = new COnOffButton(buttonRect, this, nTag,
                                           pBitmap);

        if(m_pBoostButton)
        {
                // --- always forget
                pBitmap->forget();

                // --- do any customizations then add to VC
                m_pBoostVC->addView(m_pBoostButton);
        }
}
```

And, we finish with the LED meter control, adding it to the view container, then adding the view container to the *frame*:

```
// --- add the LED VU Meter origin(25,30) size(25, 35)
/*
        Preferred Constructor:
        CVuMeter(const CRect& size,
                        CBitmap* onBitmap,
                        CBitmap* offBitmap,
                        int32_t nbLed,
                        int32_t style = kVertical);

        The VU Meter object is one of several VSTGUI4 objects that
        requires 2 bitmaps (CSlider is another), in this case we
        need one bitmap for the ON state and another for the OFF
        state. You also need to know the number of LEDs in the me
        ter.

        Our built-in LED bitmaps are only for vertical orientation,
        but it is easy to generate graphics for horizontal meters.
        Note the style constant.

        NOTE: VU meters must be manually updated in the idle()
                function to animate them!
*/
```

Copyright © 2015 Will Pirkle

```
            CPoint meterOrigin(70, 10);
            CPoint meterSize(17, 86);
            CRect meterRect(meterOrigin, meterSize);
            nTag = 50; // RAFX ControlID for Meter 1

            // --- set background bitmap, color, or use setTransparency for
            //     transparent VC
             CBitmap* onBitmap = getBitmap("vuledon.png");
             CBitmap* offBitmap = getBitmap("vuledoff.png");

             if(onBitmap && offBitmap)
             {

                  // --- create: 20 is the number of LED segments
                  m_pLeftMeter = new CVuMeter(meterRect, onBitmap, offBitmap,
                                              20, kVertical);
                  if(m_pLeftMeter)
                  {
                        // --- set the tag; note that this is not really
                        //     needed since
                        //     we need to update meters manually in idle(),
                        //     but if you have multiple meters, you may want
                        //     to index them for your own bookkeeping
                        m_pLeftMeter->setTag(nTag);

                        // --- forget
                        onBitmap->forget();
                        offBitmap->forget();

                        // --- do any customizations then add to VC
                        m_pBoostVC->addView(m_pLeftMeter);
                  }
             }

            // --- add VC to the frame
            frame->addView(m_pBoostVC);
      }

      // --- activate
      frame->onActivate(true);
}
```

Now that the controls are instantiated, we need to initialize them using the values from our plug-in buddy object. Here we use more of the built-in helper functions to make this as painless as possible. The first part of the function deals with setting the AU listener. This is for Audio Unit plug-ins only. It is important to connect as a listener so we can update the controls when the user selects a preset from the AU host. I have removed that code here, but you are free to examine it and debug it to see what is going on in AU.

```
void CVSTGUIController::initControls(bool bSetListener)
{
      if(!frame)
            return;

#ifdef AUPLUGIN
      if(m_AUInstance && bSetListener)
      {
            <SNIP SNIP SNIP>
      }
```

Copyright © 2015 Will Pirkle

```
#endif
```

Now we initialize the controls using the control value in the RackAFX *CUICtrl* object that corresponds to the Control ID value we used to to setup each control. The sequence is basically the same for each control:

- use *getUICtrlByControlID()* to get the CUICtrl for a given Control ID value
- use *getNormalizedValue()* to get the normalized version of the underlying RackAFX variable for that *CUICtrl* object
- use *setValue()* on the control object to set it
- if the control is a *CTextEdit*, use the helper function *setEditControlValue()* to set it — this function converts the value into a string for this control to accept

We'll start with the Volume Left control, which has a knob and edit box that share the same control-tag:

```
	// Get the control object with the ID value: Volume (L) = 8
	CUICtrl* pUICtrl = m_pPlugIn->getUICtrlByControlID(8);
	if(pUICtrl)
	{
		// --- set the initial value; use built in helper functions to
		//     make this simple
		float fNormalizedValue = getNormalizedValue(pUICtrl);

		// --- set it on controls with this tag/ID value
		m_pVolLeftKnob->setValue(fNormalizedValue);

		// --- edit controls are trickier - need an extra function to
		//     make this easier
		setEditControlValue(m_pVolLeftEdit, pUICtrl);
	}
```

Now, we'll do the Boost button:

```
	// --- BOOST control
	pUICtrl = m_pPlugIn->getUICtrlByControlID(45);
	if(pUICtrl)
	{
		// --- set the initial value; use built in helper functions to
		//     make this simple
		float fNormalizedValue = getNormalizedValue(pUICtrl);

		// --- set it on controls with this tag/ID value
		m_pBoostButton->setValue(fNormalizedValue);
	}
```

There is no initialization for out LED meter object — the reason is that its value is always updated in the *idle()* method so there is really no reason to repeat that here. We finish off the function by calling the *invalid()* method on the *frame* object to force the controls to repaint themselves.

```
	// --- call the repaint() function on frame
	frame->invalid();
}
```

valueChanged()

This is the last real function we need to cover (the remaining two helper functions are fairly self explanatory). This function is called when the user changes a control. The argument is the *CControl** of the object

Copyright © 2015 Will Pirkle

that was altered. We use the object's control-tag value to lookup the RackAFX CUICtrl object that holds the underlying variable and then modify it.

What makes this function different is that we are receiving a normalized value that needs to be converted to the plain value our plug-in expects. We need to handle the cases of volt/octave and log controls that are available in RackAFX. Another issue is if the changed control is a *CTextEdit* object. In this case there is more work to do to translate the string into a value we can use. Again, I have given you helper functions to handle those details, so the coding is much simpler. In the first part of the function, we do some validation and look up the *CUICtrl* object for this control-tag and validate the pointer:

```
void CVSTGUIController::valueChanged(VSTGUI::CControl* pControl)
{
	if(!m_pPlugIn) return;

	// --- get the RAFX ID for this control
	int32_t nTag = pControl->getTag();

	// --- get the control for re-broadcast of some types
	CUICtrl* pUICtrl = m_pPlugIn->getUICtrlByControlID(nTag);
	if(!pUICtrl) return;
```

Next, we will will retrieve the normalize value from the control, testing first to see if the control is a *CTextEdit*, then using the appropriate function:

```
	// --- Normalized control value
	float fControlValue = 0.0;

	// --- edit controls are handled differently than all others since they
	//     are text based
	//
	// Use dynamic casting to see if this is an edit control
	CTextEdit* control = dynamic_cast<CTextEdit*>(pControl);
	if(control)
		fControlValue = updateEditControl(pControl, pUICtrl);
	else
		fControlValue = pControl->getValue();
```

Now, we need to take care of a detail regarding the VSTGUI4 objects. The *COptionMenu*, *CVerticalSwitch* and *CHorizontalSwitch* controls transmit and store integer index values rather than normalized values. The helper function *getPluginParameterValue()* checks and converts the values accordingly. You can check the function for yourself to see how that works.

```
	// --- this function handles the case of Option Menus, which are a bit
	//     different as they store actual, not normalized, index values
	float fPluginValue = getPluginParameterValue(fControlValue, pControl);
```

Now that we have the normalized plug-in value, we need to alter it if the control is volt/octave or log based, again using the helper functions I've supplied:

```
	// --- deal with log/volt-octave controls
	if(pUICtrl->bLogSlider)
		fPluginValue = calcLogPluginValue(fPluginValue);
	else if(pUICtrl->bExpSlider)
		fPluginValue = calcVoltOctavePluginValue(fPluginValue, pUICtrl);
```

Copyright © 2015 Will Pirkle

At this point, the *fPluginValue* variable has the final, proper normalized value. We then use the last helper function *setPlugInParameterNormalized()* to convert the normalized value into the plain plug-in value, set it on the *CUICtrl* object, and call the *userInterfaceChange()* function all together:

```
// --- fPluginValue is now final normalized value for plugin
//
// --- this helper function also calls userInterfaceChange()
setPlugInParameterNormalized(pUICtrl, fPluginValue);
```

Before we can celebrate, we have another detail to handle — we need to broadcast this control change to all the **other** controls that share the same control-tag. Here only the Volume Left knob and edit control share a control-tag so that is the only case we need to handle. Notice that we need to manually invalidate the knob to redraw it if the user adjusts the value in the edit control (or any other control with the same tag). We don't need to invalidate the edit control; *setEditControlValue()* will set the text which automatically invalidates the control for redrawing.

```
	// --- now broadcast control change to all other controls with same
	//     tag, but not this control
	switch(nTag)
	{
		case 8: // Volume (L)
		{
			// --- all controls will use setValue()
			if(pControl != m_pVolLeftKnob)
			{
				m_pVolLeftKnob->setValue(fPluginValue);
				m_pVolLeftKnob->invalid(); // redraw it!
			}

			// --- EXCEPT for the edit controls
			setEditControlValue(m_pVolLeftEdit, pUICtrl);

			break;
		}

		case 45: // boost
		{
			// --- nothing to do, there is only one boost
			break;
		}
	}
}
```

Whew! That's it — we're done! The final functions *getBitmap()* and *getNormalizedValue()* in the .cpp file and the other miscellaneous helper functions in the .h file are self-explanatory and you can peruse them at your own pace.

Compile and Test

Now, compile and test the plug-in. First, make sure the sendUpdateGUI() code is working by using the Update GUI button on the RackAFX UI. Next, you are going to notice that the LED meter object does not behave like the RackAFX meter objects (either on the main view, or in the RackAFX custom GUIs). It seems to be glitchy and does not move smoothly. This is because the RackAFX meters are customized versions. They include a *CEnvelopeDetector* object inside them which allows you to control how the LED meter tracks the signal as linear/log as well as letting you set the attack and release times. The stock CVuMeter only displays the passing value and has no ballistics to it. So, it looks glitchy. If you want a simple challenge, subclass the *CVuMeter* object to include an envelope detector.

Copyright © 2015 Will Pirkle

References:

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