

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.

StealthDelay by PolygonSheep (created with this GUI Tutorial and RackAFX v6.8):
<http://www.polygonsheep.com/>



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



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



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 faint 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.

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=cp3draeYLP>

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

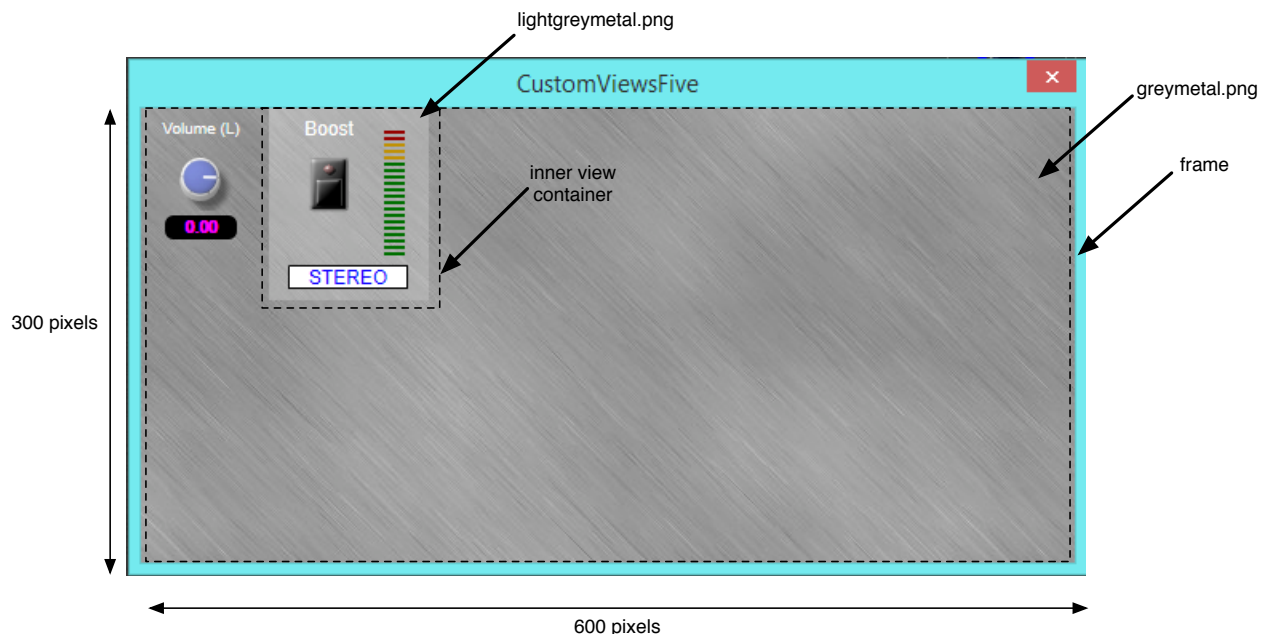
1. copy the .png file into the <project>/resources folder
2. in the Visual Studio Solution Explorer, find the <project>.rc file and right click on it; choose “View Code” and add a line to your .rc file like this (notice that the all CAPS name is identical to the lower case version):

```
////////////////////////////////////  
//  
// 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 or Make AAX. 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**:



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, option menu 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.

VSTGUI Control Tags vs. RackAFX ControlIDs

In RackAFX, your controls are identified by a unique identifier called the ControlID. When you set up controls on the main UI with the slider and button designer, those ControlID values are hardcoded for you and can not be changed. Calls to *userInterfaceChange()* and *checkUpdateGUI()* in RackAFX also use these ControlID values. In VST2, the GUI controls were also referenced with an ID value, but these had to be zero-indexed and in sequential order. VSTGUI4 uses the *controlTag* as the identifier. Historically, this was a zero-indexed value that matched the value in the VST2 plugin.

The UI control objects in your RackAFX plugin are stored in a linked list that may be accessed using the list's index value (zero-indexed as in the VST2 plugins) or with the RackAFX ControlID value - see the *plugin.h* file for those access functions. During the ordinary course of operations, you only need to use the ControlID value to access your UI objects.

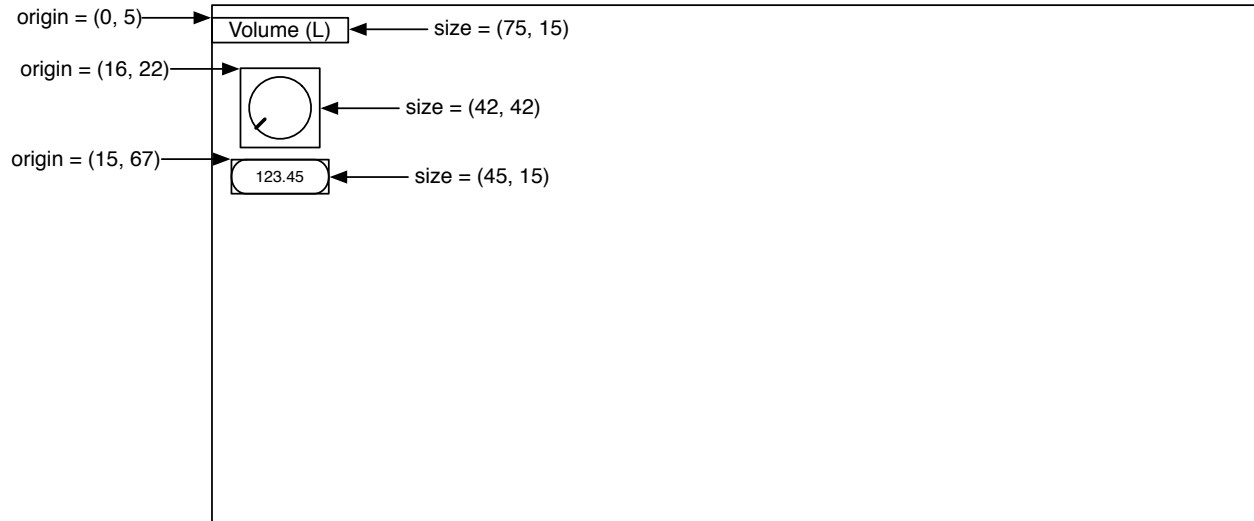
However, when designing the base class object for this project, in order to keep the ID values consistent across AU, VST and AAX, we need to reference the controls with their zero-indexed controlTag value. The included base class object contains functions to help you convert your RackAFX ControlID values to and from the zero-indexed controlTag values. For example:

```
int nTag = getTagForRAFXControlId(8);
```

gets the tag version of the GUI control with RackAFX ControlID = 8. As you work through this tutorial, just remember to keep that relationship in mind.

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.

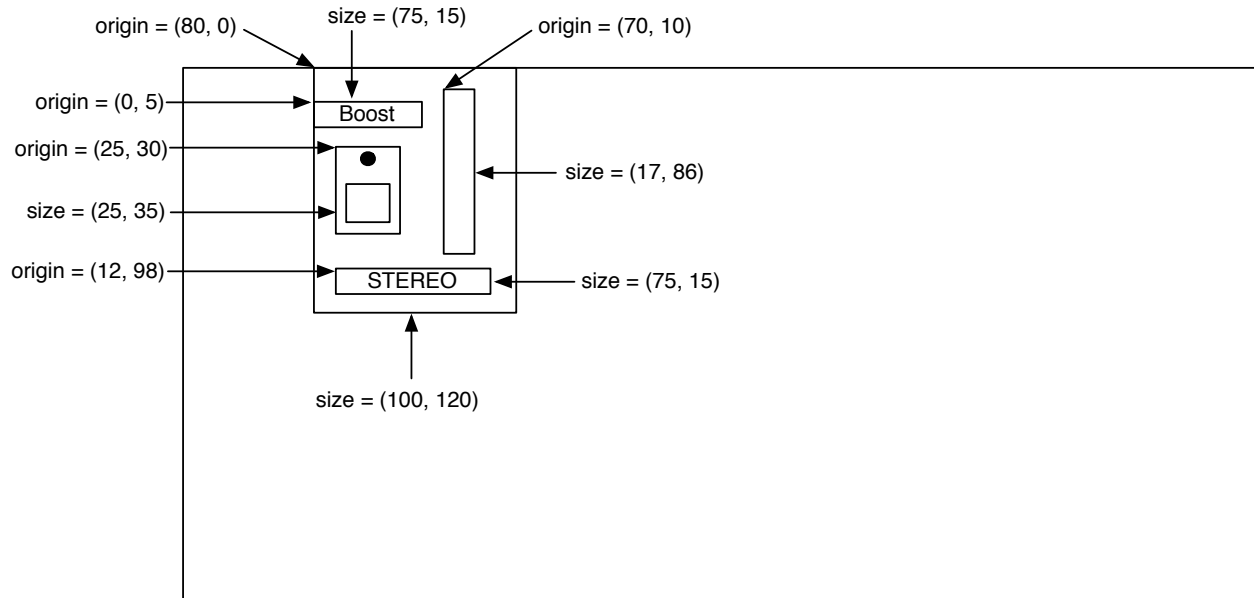


Notice that the coordinates of the origin values are relative to the *frame*, whose upper left corner is (0, 0). This is the golden rule of VSTGUI object placement — the origin of the object is always relative to the origin of the parent container. Also notice that the x-coordinate increases as you move from the left to right and the y-coordinate increases as you move down from the top - a common UI paradigm. So, we can now list our first three GUI objects that will need to be created and their origin/size values:

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
COptionsMenu	(12, 98)	(75, 15)	n/a

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

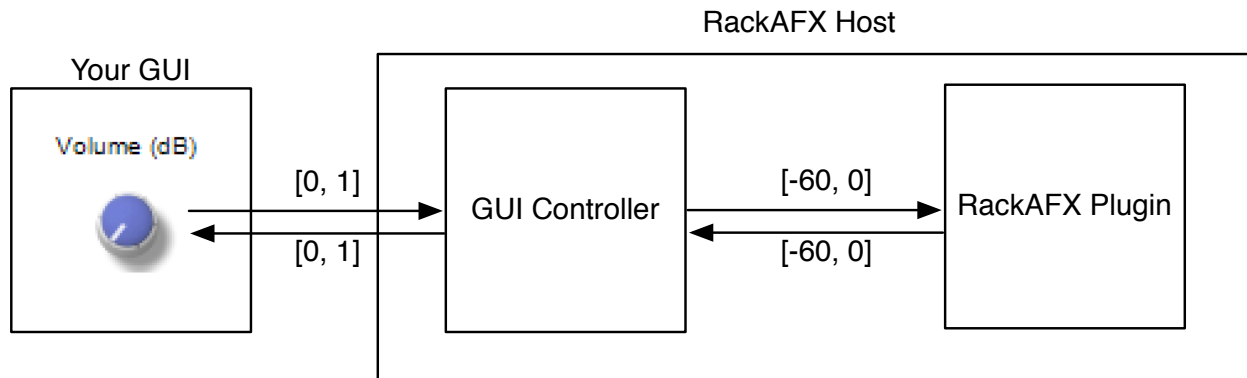
- create the *COnOffButton* and add it to the inner view container as a sub-view
- create the *CVuMeter* and add it to the inner view container as a sub-view
- create the *COptionMenu* and add it to the inner view container as a sub-view

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

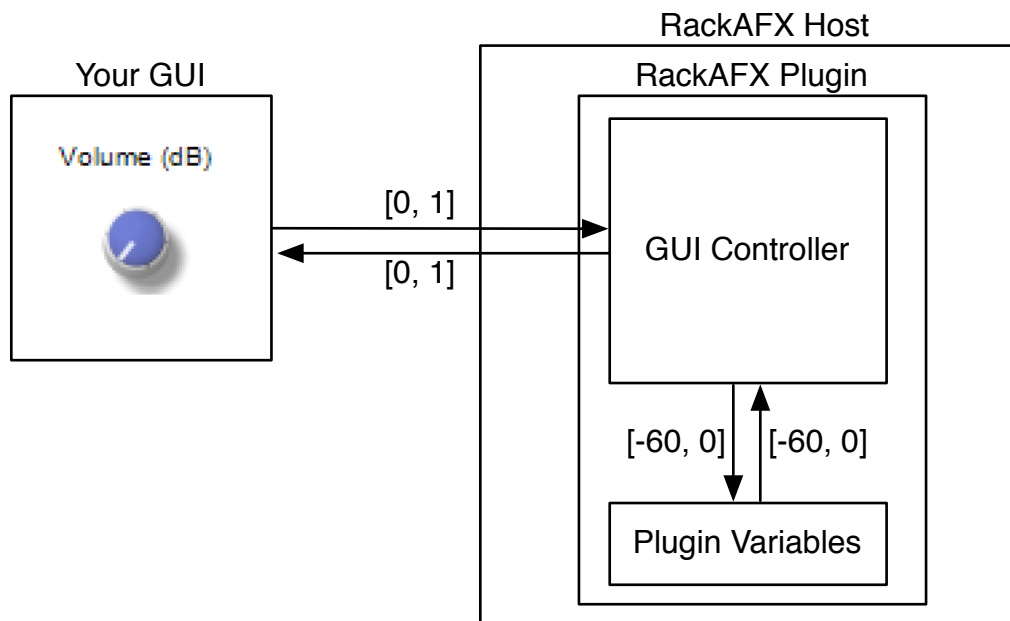
The GUI Editor Object

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:



Thread-safe GUI Communication

Starting in RackAFX v6.8, all GUI communications in native RackAFX as well as all the ported projects (Make VST, Make AU, Make AAX) are handled in a thread-safe manner. This is documented on my website at <http://www.willpirkle.com/forum/> and is too detailed to cover here. The main thing to understand is that each of the APIs has its own way of ensuring that control changes are applied to the plugin prior to audio processing. In this way, controls will not change during the audio buffer processing loop and the accesses to shared data via multiple threads is done in a manner that is thread-safe.

In all APIs, an intermediate container of parameter values, objects or structures is also applied. These are:

RackAFX: *GUI_PARAMETER* structures
VST3: *Parameter* objects
AU: *GlobalParameter* objects
AAX: *AAX_IParameter* objects

VST3, AU and AAX also require these additional objects for implementing host-drawn GUI's (also known as "Default GUIs"), preset loading/storing, and GUI communications. We need to ensure that GUI changes on the Custom GUI are also properly applied to any API-specific underlying parameter objects. In VST3 this is called "parameter binding" and is a good name for this paradigm.

The Good News!

The good news is that I've already written all the code that will synchronize your Custom GUI parameter values to the appropriate API-specific objects and structures. You do not need to worry about that part or modify any of the underlying code. These details are documented in the code and you can also find more information from my website.

CVSTGUIController

I've created this abstract base class for you to subclass with your own Custom GUI object. You will need to implement only a few functions to get your GUI working properly. Other functions are written for you but you will not want or need to modify that code. In this document, we will focus on the functions that you **do** need to implement.

CVSTGUIController is derived 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.

```
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;
};
```

IControlListener

The GUI objects will deliver control change information to their *IControlListener* buddies. In the previous modules, the listener was the RackAFX GUI (or VST3/AU/AAX wrapper objects). Now, your object must handle these chores. *IControlListener* 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 IControlListener
{
public:
    virtual ~IControlListener() {}
    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) {}
#ifdef 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
- find the CUICtrl object that is connected to this control-tag

- translate the normalized value from the control to the required value for the plug-in
- call the built-in function `setPluginParameterNormalized()` which will handle the API-specific thread-safe handling of transferring this value into the plugin
- broadcast this control change to all GUI objects that have the same control-tag by calling `setGUIControlsWithNormalizedParameter()` which is one of the few functions you must write

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's abstract base class. 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, VST3 and AAX 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 `IControlListener`

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.

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 and parameter changes are handled in a thread-safe manner. 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. Here is the declaration of the *CVSTGUIController* showing only the functions you need to modify to make your Custom GUI work properly. It does not show the helper functions (for doing things like creating PNG images) or the API Specific 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 {

// --- the main VSTGUI object, platform independent
class CVSTGUIController : public VSTGUIEditorInterface,
                          public IControlListener,
                          public CBaseObject
{
public:
    CVSTGUIController();
    virtual ~CVSTGUIController();

    // --- MAIN OVERRIDES:
    //
    // --- open()
    virtual bool open(CPlugin* pPlugin, VSTGUI_VIEW_INFO* info);

    // --- close function
    virtual void close();

    // --- do idle processing; optional override for meters
    virtual void idle();

    // --- VSTGUIEditorInterface override; you must override this if you
    // want to alter the knob mode
```

```
    virtual int32_t getKnobMode() const;

    // --- IControlListener override (pure abstract, so you **must** over
    //     ride this
    virtual void valueChanged(VSTGUI::CControl* pControl) = 0;

    // --- you must override this functions to set your control locations
    //     based on the normalized values
    virtual void setGUIControlsWithNormalizedParameter(int tag,
                                                       double value,
                                                       VSTGUI::CControl* pControl = NULL) = 0;

protected:
    void* m_hPlugInInstance; // HINSTANCE of this DLL

    // --- our plugin
    CPlugIn* m_pPlugIn;

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

CCustomVSTGUI

In the accompanying project, I've subclassed CVSTGUIController as CCustomVSTGUI:

```
class CCustomVSTGUI : public CVSTGUIController
```

It includes only nine functions, and you do not need to modify two of them, so there are really only seven functions to deal with; a few of them are very simple.

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 *CVSTGUIController* object:

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

// our custom GUI object
#include "CustomVSTGUI.h"

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: ----- //
    CCustomVSTGUI* 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
- respond to GUI_RAFX_SYNC message by calling the *initControls()* method, which will sync all controls to their current values, used for loading presets
- respond to GUI_EXTERNAL_SET_NORM_VALUE message by calling the appropriate function on the custom GUI object; only used for parameter binding in VST3

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  
    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 CCustomVSTGUI;
    if(m_pGUIController)
    {
        m_pGUIController->open(this, info);
    }
    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;
    }
    // returning NULL = success
    return m_pGUIController;
}

// --- for any All-at-once sync (RAFX or others)
case GUI_RAFX_SYNC:
{
    if(m_pGUIController)
    {
        m_pGUIController->initControls();
    }
    return NULL;
}

// --- for sync with VST parameters only,
//      and only for supporting pure Custom GUIs
case GUI_EXTERNAL_SET_NORM_VALUE:
{
    if(m_pGUIController)
    {
        m_pGUIController>setGUIControlsWithNormalizedParameter(
            info->parameterTag,
            info->normalizedParameterValue);
    }
    return NULL;
}

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

```
        {  
            return NULL;  
        }  
    }  
  
    return NULL;  
}
```

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.

CCustomVSTGUI.h

Below is the complete declaration of our derived object. The functions that you need to work with are in **bold** along with the member variable pointers to all the VSTGUI objects.

```
#pragma once  
#ifndef __CUSTOMVSTGUI_H  
#define __CUSTOMVSTGUI_H  
  
#include "VSTGUIController.h"  
  
class CCustomVSTGUI : public CVSTGUIController  
{  
public:  
    CCustomVSTGUI(void);  
    ~CCustomVSTGUI(void);  
  
    // --- overrides  
    virtual bool open(CPlugIn* pPlugIn, VSTGUI_VIEW_INFO* info);  
  
    // --- do idle processing; meters  
    virtual void idle();  
  
    // --- pure abstract base class so we **must** override this  
    virtual void valueChanged(VSTGUI::CControl* pControl);  
  
    // --- must override for parameter sync across various APIs  
    //     the implementation will depend on your custom GUI code  
    //     but there are helper functions below to make this as easy as  
    //     possible  
    virtual void setGUIControlsWithNormalizedParameter(int tag,  
                                                       double normalizedValue,  
                                                       VSTGUI::CControl* pControl = NULL);  
  
    // --- create controls: common to all APIs  
    void createControls();  
  
    // --- init controls: varies with API  
    void initControls();
```



```

// --- knob mode override
virtual int32_t getKnobMode() const;

// --- threadsafe version for GUI updates from plugin
void checkSendUpdateGUI(int tag, float fValue,
                        VSTGUI::CControl* pControl);

// --- set RAFX parameter via different APIs
float setPlugInParameterNormalized(int nTag, float fNormalizedValue,
                                  VSTGUI::CControl* pControl);

protected:
// --- GUI SPECIFIC CONTROLS
//     since this is a simple example, I will declare them
//     individually, but you could also declare as arrays of controls
CTextLabel*      m_pVolLeftLabel;
CAnimKnob*       m_pVolLeftKnob;
CTextEdit*       m_pVolLeftEdit;
CViewContainer*  m_pBoostVC;
CTextLabel*      m_pBoostLabel;
COnOffButton*    m_pBoostButton;
CVuMeter*        m_pLeftMeter;
COptionsMenu*    m_pChannelOptionsMenu;
};
#endif

```

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 pre-set.

CCustomVSTGUI.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

```

CCustomVSTGUI::CCustomVSTGUI ()
{
    m_pVolLeftLabel = NULL;
    m_pVolLeftKnob = NULL;
    m_pVolLeftEdit = NULL;

    m_pBoostVC = NULL;
    m_pBoostLabel = NULL;
    m_pBoostButton = NULL;
    m_pLeftMeter = NULL;
    m_pChannelOptionsMenu = NULL;
}

```

Destructor:

- nothing to do here

```

CCustomVSTGUI::~CCustomVSTGUI ()
{

```

```
}
```

getKnobMode()

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

```
int32_t CCustomVSTGUI::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:

```
virtual bool open(CPlugIn* pPlugIn, VSTGUI_VIEW_INFO* info);
```

VSTGUI_VIEW_INFO* info

This structure contains all the necessary information for creating the GUI, including some API-specific variables. We will discuss these as needed. Importantly, the structure contains the two return variables from this function - the GUI's width and height.

CPlugIn* pPlugIn

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

The **open()** function needs to do the following:

- call the base class function to setup all the API-specific stuff for us; this also stores our plugin pointer
- set the plug-in instance variable (WinOS only)
- set the AU Listener mechanism (AU only) - you do not need to modify this code or deal with it at all
- set the dimensions on the return arguments
- create the *frame* object
- call the *open()* method on the *frame* object and pass it the window pointer that was given in the info structure
- set the *frame*'s background color and/or bitmap graphic
- populate the *frame* with GUI controls
- initialize the controls

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 CCustomVSTGUI::open(CPlugIn* pPlugIn, VSTGUI_VIEW_INFO* info)
{
    // --- first call the base class to store the API-specific stuff; it
    //       will handle most of these chores for you
    bool bSuccess = CVSTGUIController::open(pPlugIn, info);

    // critical failure at base class level, should never happen
    if(!bSuccess) return false;
}
```

```
#ifndef AUPLUGIN
    if(m_AUInstance)
    {
        // --- create the event listener and tell it the name of our Dis
        //      patcher function named:
        //      EventListenerDispatcher
        verify_noerr(AUEventListenerCreate(EventListenerDispatch, this,
                                          CFRunLoopGetCurrent(),
                                          kCFRunLoopDefaultMode, 0.05, 0.05,
                                          &m_AUEventListener));

        // --- start with first control 0
        AudioUnitEvent auEvent;

        // --- parameter 0
        AudioUnitParameter parameter = {m_AUInstance, 0,
                                         kAudioUnitScope_Global, 0};

        // --- set param & add it
        auEvent.mArgument.mParameter = parameter;
        auEvent.mEventType = kAudioUnitEvent_ParameterValueChange;

        verify_noerr(AUEventListenerAddEventType(m_AUEventListener, this,
                                                &auEvent));

        // --- notice the way additional params are added using mParameterID
        for(int i=1; i<m_pPlugIn->getControlCount(); i++)
        {
            auEvent.mArgument.mParameter.mParameterID = i;
            verify_noerr(AUEventListenerAddEventType(m_AUEventListener,
                                                    this, &auEvent));
        }
    }
#endif

// --- set the return variables (you may want to store them too)
info->size.width = 600;
info->size.height = 300;
```

- create the *frame* object
- set the *frame's* background color; two examples are shown here, the first uses built-in colors and the second constructs the color from RGBA

```
// --- create the frame rect: it dictates the size in pixels
CRect frameSize(0, 0, info->size.width, info->size.height);

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

// --- open it
#if defined _WINDOWS || defined WINDOWS || defined _WINDLL
frame->open(info->window, kHWND); // for WinOS, window = HWND
#else
// for MacOS, window = NSView* or HUIView* (VST2 wrapper only)
frame->open(info->window, (PlatformType)info->vstPlatformType);
#endif

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

```

//
// COLORS: use either built-in colors, or construct your own from
//         r,g,b,a
// --- 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. The arguments ...0, 0, 0, 0) in the getBitmap() method are the tiling offsets. When set to 0, 0, 0, 0 the graphic will be tiled infinitely in the X and Y dimensions. See the VSTGUI4 documentation for more information on tiled 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 () ;

```

```

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

    return true;
}

```

idle()

This is actually one of the most important functions we need to implement. It is responsible for repainting 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). There are separate code chunks for each API, except VST3 which handles the repainting inherently. The code is simple - in each case, we are calling *setValueNormalized()* on the left meter object according to its current value in the plugin, then calling the *invalid()* method to repaint the meter. These meter values are thread-safely accessed with their own member functions, which you do not need to write or alter. The final call is for the base class to do its updates. The reason that the `m_pLeftMeter->invalid()` function call is repeated for each API is that we don't want to call `invalid()` unless we know we need it. For VST3 plugins, this function is not used so we don't want to blindly call `invalid()` for that case.

```

void CCustomVSTGUI::idle()
{
    // --- left meter control Id = 50; getRAFXParameterNormalized() takes the
    //      index (tag) so use helper function to get it
    int nTag = getTagForRAFXControlId(50);

    // --- RAFX: get the RAFX parameter to set the meter value
    #if !defined AUPLUGIN && !defined AAXPLUGIN && !defined VSTPLUGIN
        m_pLeftMeter->setValueNormalized(getRAFXParameterNormalized(nTag));
        m_pLeftMeter->invalid();
    #endif

    // --- AU: get the AU parameter to set the meter value
    #ifdef AUPLUGIN
        m_pLeftMeter->setValueNormalized(getAUPParameterActual(nTag));
        m_pLeftMeter->invalid();
    #endif

    // --- AAX: get the RAFX parameter to set the meter value
    #ifdef AAXPLUGIN
        m_pLeftMeter->setValueNormalized(getAAXParameterNormalized(nTag));
        m_pLeftMeter->invalid();
    #endif

    // --- VST: not needed; this is passed back as a param update from the
    //      process function and connected directly to the meter update

    // --- let base class do its thing
    CVSTGUIController::idle();
}

```

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, let's 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 CCustomVSTGUI::createControls ()
{
    if(!frame)
        return;

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

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 using 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 — it is the Control ID in RackAFX but converted to the controlTag version with the helper function. **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);

// --- v6.8+ --> the VSTGUI4 Control Tag MUST match the index of the
//               control in our control list
//               for proper parameter synchronization with the VST, AU,
//               AAX or RAFX Host
//
// 8 = left Volume ControlId in RackAFX; function returns index of con-
//     trol, sync'd to VST/AU/AAX parameters
int nTag = getTagForRAFXControlId(8);
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,
           IControlListener* 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 derived 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)
    {
        // --- 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) drop-list and a LED VU Meter inside it.
*/

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

// --- get the proper VSTGUI tag
int nTag = getTagForRAFXControlId(45); // 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);
    }
}
```

Next is the option menu (drop-list) which I added here to show how to access the enumerated list from the RAFX control object. This is one of the few controls that requires extra initialization at construction time:

```
// --- add the COptionMenu origin(25,30) size(25, 35)
/*
    Preferred Constructor:
    COptionMenu(const CRect& size,
                IControlListener* listener,
                int32_t tag,
                CBitmap* background = 0,
                CBitmap* bgWhenClick = 0,
                const int32_t style = 0);
*/
CPoint optionMenuOrigin(12, 98);
CPoint optionMenuSize(75, 15);
CRect optionMenuRect(optionMenuOrigin, optionMenuSize);

// --- get the proper VSTGUI tag
// 41 = RAFX ControlID for Channel selector
nTag = getTagForRAFXControlId(41);

// --- create button
m_pChannelOptionMenu = new COptionMenu(optionMenuRect, this,
                                       nTag);

// --- no bitmap required, but can be used for custom menu
if(m_pChannelOptionMenu)
{
    // --- do any customizations then add to VC
    //     let's make the background white and text blue
    m_pChannelOptionMenu->setBackColor(kWhiteCColor);
    m_pChannelOptionMenu->setFontColor(kBlueCColor);

    // --- populate the option menu with strings from the con-
    //     nected CUICtrl object
    //     This is one of the VSTGUI4 controls that requires
    //     additional initialization
    //     CSegmentButton is another one
    // --- get the object with Control ID = to the Option Menu
    //     (set in CreateControls() above)
    CUICtrl* pUICtrl = m_pPlugIn->getUICtrlByListIndex(
                                       m_pChannelOptionMenu->getTag());
if(pUICtrl)
{
    // --- clear
    m_pChannelOptionMenu->removeAllEntry();

    // --- check to make sure this is a UINT control
    //     (this is an extra step)
    if(pUICtrl->uUserDataTypes == UINTData)
{
        // --- iterate through the cEnumeratedList
        //     until there is nothing left
        bool bWorking = true;
        int m = 0;
        while(bWorking)

```

```
{
    // --- chew off the next string
    stringstream enumString;
    getEnumString(pUICtrl->cEnumeratedList,
                 m, enumString);

    // --- if empty, we are done
    if(enumString.str().size() == 0)
        bWorking = false;
    else
    {
        // --- add this string to the menu
        m_pChannelOptionsMenu->addEntry(
            enumString.str().c_str(), m);

        m++;
    }
}

// --- make sure there are entries; if the
//     above failed (it should never), then set
//     it to something to avoid a crash
int n = m_pChannelOptionsMenu->getNbEntries();
if(n <= 0)
{
    m_pChannelOptionsMenu->addEntry("-n/a-");
    m_pChannelOptionsMenu->setMax(0);
    m_pChannelOptionsMenu->setValue(0);
}
}

// --- add it
m_pBoostVC->addView(m_pChannelOptionsMenu);
}
```

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 meter.

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!
*/
CPoint meterOrigin(70, 10);
CPoint meterSize(17, 86);
CRect meterRect(meterOrigin, meterSize);
nTag = getTagForRAFXControlId(50); // 50 = 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 in our API's particular parameter container. This is only needed for RackAFX and AU; the others are self-synchronizing.

```
void CCustomVSTGUI::initControls()
{
    if(!frame)
        return;

    // --- Synchronization to plugin parameters, API dependent
    //
    int nParams = m_pPlugIn->getControlCount();
    for(int i=0; i<nParams; i++)
```

```
{
    #ifndef AUPLUGIN
        // --- For AU plugins, we sync to the ACTUAL parameters stored in
        //      our Globals
        setGUIControlsWithActualParameter(i, getAUPParameterActual(i));
    #endif

    #if !defined AUPLUGIN && !defined AAXPLUGIN && !defined VSTPLUGIN
        // --- For RAFX plugins, we sync to the NORMALIZED parameters
        //      stored in our Globals
        setGUIControlsWithNormalizedParameter(i,
                                              getRAFXParameterNormalized(i));
    #endif
}

// --- For VST and AAX plugins there is *nothing to do* -- the params
//      are initialized automatically after the GUI is shown
//      via the setParameterNormalized( ) function below, this is part
//      of the API for each

// --- force the whole GUI to repaint
frame->invalid();
}
```

setGUIControlsWithNormalizedParameter()

This control will be called for one of two reasons:

- (1) externally called to sync our GUI with the VST client GUI (Ableton allows you to see both at once), *pControl* will be NULL
- (2) internally called to sync all controls with the same tag to the same value for RAFX only, here *pControl* will NOT be NULL and can be used to check if the control really needs updating (an infinite loop won't happen, but I like to play it safer here)

You will need to modify this code for each of your control objects that can be updated (for example, static text does not need updating). The operation is simple here - use the incoming *tag* and *pControl* values to figure out which of your control object needs to be set, then call *setValueNormalized()* for non-text based controls, or *updateTextControl()* for text-based controls (edit boxes or static text that you wish to change dynamically). Notice the way the inputs are used to find the correct control:

```
void CCustomVSTGUI::setGUIControlsWithNormalizedParameter(int tag,
                                                         double normalizedValue,
                                                         VSTGUI::CControl* pControl)
{
    if(m_pVolLeftKnob && m_pVolLeftKnob != pControl
        && m_pVolLeftKnob->getTag() == tag)
    {
        m_pVolLeftKnob->setValueNormalized(normalizedValue);
        m_pVolLeftKnob->invalid();
    }

    if(m_pChannelOptionsMenu && m_pChannelOptionsMenu != pControl
        && m_pChannelOptionsMenu->getTag() == tag)
    {
        m_pChannelOptionsMenu->setValueNormalized(normalizedValue);
        m_pChannelOptionsMenu->invalid();
    }

    if(m_pBoostButton && m_pBoostButton != pControl
```

```
        && m_pBoostButton->getTag() == tag)
    {
        m_pBoostButton->setValueNormalized(normalizedValue);
        m_pBoostButton->invalid();
    }

    if(m_pVolLeftEdit && m_pVolLeftEdit != pControl
        && m_pVolLeftEdit->getTag() == tag)
    {
        updateTextControl(m_pVolLeftEdit, tag, normalizedValue);
        m_pVolLeftEdit->invalid();
    }

    if(m_pLeftMeter && m_pLeftMeter->getTag() == tag)
    {
        m_pLeftMeter->setValueNormalized(normalizedValue);
        m_pLeftMeter->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 that was altered. In this function we will need to:

- get the RAFX *CUICtrl* pointer with the control's *tag* variable
- get the normalized value from the control; this is done differently for text-based controls so we first do a test to see if the control is text-based then use the appropriate built-in function
- use the built-in function *setPluginParameterNormalized()* to handle the thread-safe accesses to the underlying API's parameters; for VST, AU and AAX, this will also sync all controls that share the same tag
- call the built-in *checkSendUpdateGUI()* function to handle user updates; there is nothing for you to alter in that function
- for RAFX, broadcast this control change to all controls with the same tag via *setGUIControlsWithNormalizedParameter()* which we implemented above

```
void CCustomVSTGUI::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->getUICtrlByListIndex(nTag);
    if(!pUICtrl) return;

    // --- Normalized control value
    float fNormalizedValue = 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)
        fNormalizedValue = parseAndUpdateTextControl(
            (CTextEdit*)pControl, pUICtrl);
    else
        fNormalizedValue = pControl->getValueNormalized();

    // --- RAFX, VST, AU, AAX done here; NOTE: for VST, AU, AAX this will
    //      update all controls with same tag
    float fWouldBeValue = setPlugInParameterNormalized(nTag,
        fNormalizedValue, pControl);

    // --- check update GUI RAFX, VST, AU, AAX
    checkSendUpdateGUI(nTag, fWouldBeValue, pControl);

    // --- RAFX: broadcast the new value to all controls with same tag
    #if !defined AUPLUGIN && !defined AAXPLUGIN && !defined VSTPLUGIN
        if(!m_pVSTParameterConnector)
            setGUIControlsWithNormalizedParameter(nTag, fNormalizedValue,
                pControl);
    #endif
}
```

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.

References:

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