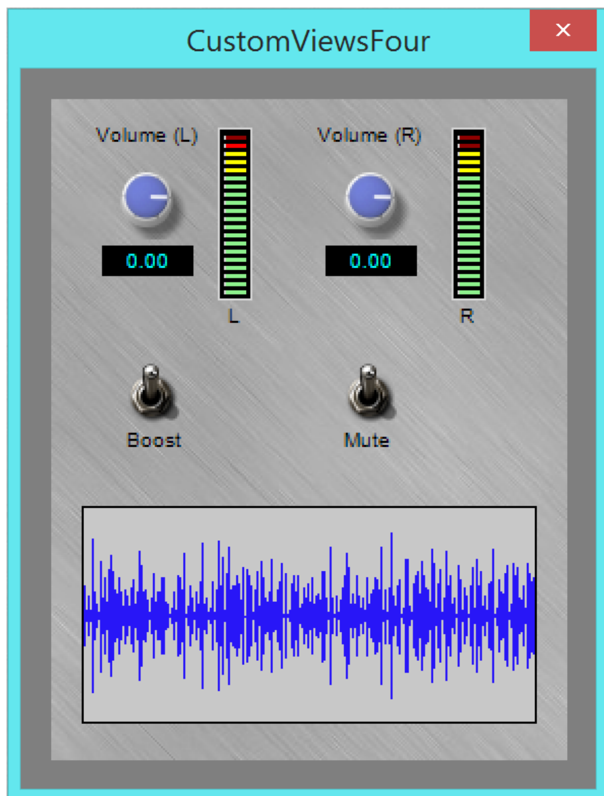
Copyright © 2015 Will Pirkle

Creating VSTGUI Custom Views 5: **Subclassing *CView* for View Objects**
Will Pirkle

In this module, we will subclass *CView*, the mother of all view objects and create a waveform graph that shows the audio waveform scrolling from left to right as audio streams. In the process we will learn about the primitive drawing functions that you can use in your *CViews* - these are the same types of functions I use in the RackAFX analyzer to plot waveforms, FFTs, etc… So you can add this kind of feature to your product with some effort. The project that accompanies this module is called **CustomViewsFour**.



The custom *CView* object is at the bottom and displays the audio waveform.

The Wave View object
In the last module, you saw the member attributes and methods for *CView*. For our scrolling waveform view we only need to override the *draw()* function. But of course there are many details about the drawing we need to handle including learning some primitive drawing functions. Examine the waveform view above and observe that:

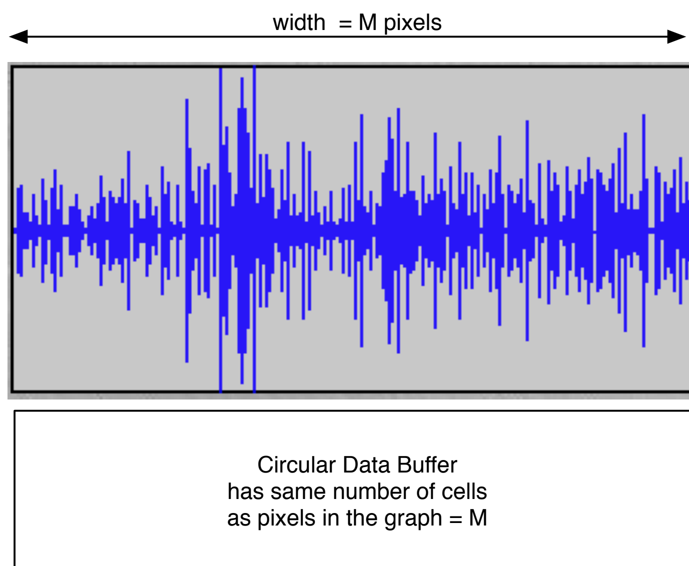
- it has a thin black outline
- it has a grey color for the background
- the audio waveform is in blue and exactly fits in the control

The audio waveform itself may look like some kind of very complex polygon that we have filled with blue color, but in actuality it is made up of only vertical lines. The places where the vertical lines touch each other looks like a filled polygon. Obviously, the height of these vertical lines is based on the audio output level. In our *draw()* function, creating the black outline and grey color fill are simple. The audio waveform is where the details lie.

Copyright © 2015 Will Pirkle

As with the previous modules, if you are attempting this advance GUI stuff then I am assuming you can debug C++ code and you are not afraid to do some work to get the results you want. So, I am not going to discuss every line of code - you need to figure out some stuff on your own to really understand and digest it.

The fundamental way this audio waveform view works is that it includes a circular buffer. The circular buffer holds the audio data amplitude information that the waveform view paints into the device context.



Here is the audio waveform view with the circular buffer below it and laying on its side. The trick to making this work is that the circular buffer has the exact same number of cells (memory locations) as the *CView* is wide, in pixels. When we plot the data, we will move one pixel at a time, drawing a line that represents the overall audio amplitude variable. Thus each cell in the buffer corresponds to one audio sample amplitude and that produces one line segment in the graph.

Each cell in the circular buffer contains the absolute value of an audio sample. Thus non-distorted values will be on the range of [0.0, 1.0] — anything above 1.0 is clipped off the graph.

The scrolling effect is done by responding to the VSTGUI_TIMER_PING message in your *showGUI()* method. In this code we will grab the next audio data point and stuff it into the circular buffer (overwriting the oldest value), then we will invalidate the wave view to force it to repaint itself. This succession of invalidation/repaints is what animates the control. In the paint code, we can move forwards or backwards through the circular buffer to make the waveform appear to scroll in either direction, though right-to-left is what you see in most DAWs, so we will use that.

The wave view object is packaged in:

WaveFormView.h - the interface file
WaveFormView.cpp - the implementation file

Copyright © 2015 Will Pirkle

Since the *CView* constructor requires a *CRect* object to define it, we will use the same kind of constructor and simply call the base class. We also need to override the *draw()* method. Finally, we need the circular buffer and its indexing variables (this is all covered in my FX book and in numerous sample projects).

CWaveFormView

Have a look at the class definition for *CWaveFormView*:

```
#pragma once
#include "../vstgui4/vstgui/vstgui.h"

namespace VSTGUI {

class CWaveFormView : public CView
{
public:
	// constructor
	CWaveFormView(const CRect& size);

	// our only override
	virtual void draw(CDrawContext *pContext);

	// stuff for circular buffer
	float* m_pCircBuffer;
	int m_nWriteIndex;
	int m_nReadIndex;
	int m_nLength;

	// methods to work on buffer
	void addWaveDataPoint(float fSample);
	void clearBuffer();
};

}
```

Notice the two methods at the bottom of the definition - one of them adds a new data point to the circular buffer - it will be called each time we respond to the VSTGUI_TIMER_PING message. The clearBuffer() below simply empties the data.

Now lets step through the methods in the implementation (.cpp) file.

Constructor:

In the constructor we need to:

- dynamically create our new circular buffer based on the width of the *CView* in pixels
- initialize the circular buffer's index and length variables
- clear out the circular buffer

Notice the items in **bold**.

```
CWaveFormView::CWaveFormView(const VSTGUI::CRect& size)
: CView(size)
{
	m_pCircBuffer = NULL;
	m_pCircBuffer = new float[(int)size.width()];
	m_nWriteIndex = 0;
```

Copyright © 2015 Will Pirkle

```
	m_nReadIndex = 0;
	m_nLength = size.width();
	m_nLength -= 1;
	memset(m_pCircBuffer, 0, m_nLength*sizeof(float));
}
```

addWaveDataPoint():
In this function we need to put a new value into the circular buffer and increment our write index variable, wrapping it around the buffer if needed. Again the good stuff is in **bold**.

```
void CWaveFormView::addWaveDataPoint(float fSample)
{
	if(!m_pCircBuffer) return;

	m_pCircBuffer[m_nWriteIndex] = fSample;

	m_nWriteIndex++;

	if(m_nWriteIndex > m_nLength - 1)
		m_nWriteIndex = 0;
}
```

clearBuffer():
In this function we just memset the buffer to 0 and reset the index values.

```
void CWaveFormView::clearBuffer()
{
	if(!m_pCircBuffer) return;

	memset(m_pCircBuffer, 0, sizeof(float));
	m_nWriteIndex = 0;
	m_nReadIndex = 0;
}
```

Primitive Drawing Functions
Before we look at the *draw()* method, lets discuss primitive drawing functions. The term “primitive” here is misleading - though these functions are all simple and basic, they can be combined to produce very nice graphic plots, especially since VSTGUI includes anti-aliasing code to make lines and curves look smooth. Remember that the entire Analyzer object in RackAFX is drawn with only the same primitive methods (they are just the Microsoft versions). With a little work, you will get used to these methods and with a little ingenuity, you can create killer graphics or even animations. The basic types of primitive drawing items are:

- lines
- arcs
- rectangles
- polygons
- ellipses (including circles)
- points
- text
- bitmaps (pre-drawn graphic components)

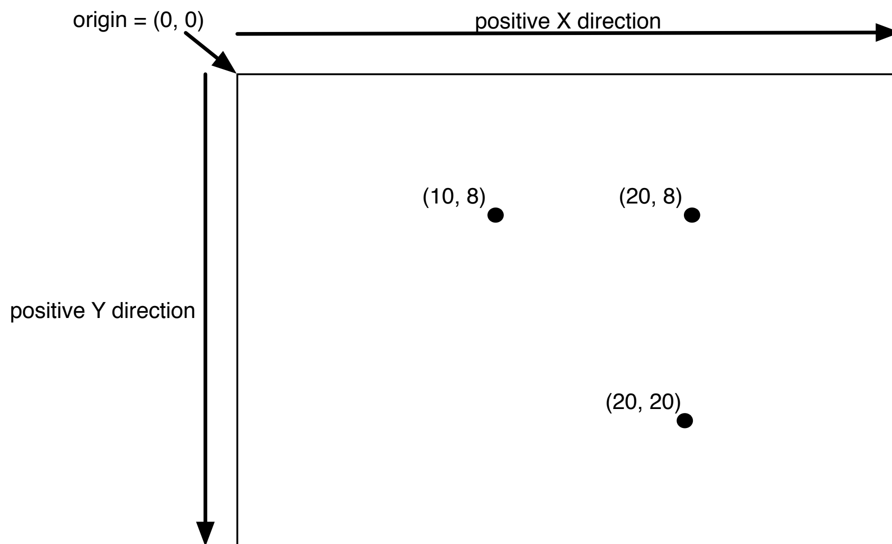
Copyright © 2015 Will Pirkle

In our wave view, we are only going to need rectangles and lines to accomplish everything we need.

GUI Coordinate System

As we discussed in module 3, the GUI coordinate system is flipped for the y-dimension. The upper left corner of the view is the origin at (0,0) and moving to the right increases x positively, while moving down increases y-positively. You need to keep this in mind when dealing with GUI drawing code.

NOTE: if you already have your hands on some code to manipulate lines, circles, planes, etc… geometrically, then you only need to do a bit of work to get them to work in this upside down world - invert the y components and make sure everything is in quadrant IV of your system.



The draw function accepts a single argument, the *CDrawContext** which renders the graphic into memory using whatever platform-dependent primitive drawing functions are needed.

```
void CWaveFormView::draw(CDrawContext* pContext)
```

Look at the class declaration for *CDrawContext* and you can see the drawing functions. Here is an abbreviated list of methods:

Method	Description
drawLine()	draw a line one CPoint to another CPoint
drawLines()	draw a series of line segments using an array of CPoint objects that set the start and end points
drawRect()	draw a rectangle using an array of CPoint objects that set the rectangle corners
drawPolygon()	draw a polygon using an array of CPoint objects that set the polygon vertices
setLineWidth()	set the width of the lines in pixels
setFillColor()	set the color for filling rectangles, polygons, ellipses or any filled shape

Copyright © 2015 Will Pirkle

Method	Description
setFrameColor()	set the color the lines (aka stroke color)
setFontColor()	set the color of the font
drawString()	draw text string

The *draw()* function components
We are going to code the draw() function to implement the following flow of logic.

First setup the background stuff - this is the static stuff behind the main graphic we will draw

- if there is a background image, draw it (we do not have one, but you can easily add your own)
- set the line width to 1 pixel
- set the fill color to light-grey (200, 200, 200, 255) in (r, g, b, a)
- set the frame (line) color to black
- call *drawRect()* to draw the rectangle according to our *size* variable and fill it with the grey color; this automatically produces both the thin black frame-line and the grey fill

Next, we need to step through the circular buffer and use the *moveTo()* and *lineTo()* functions to move and draw lines. We will make another call to change the frame (line) color to blue before drawing the lines. The logic I chose is simple, though you could also do the same thing with slightly different code.

- step through the circular buffer starting at the first cell and moving **backwards** through the buffer, which will cause the waveform to move from left to right; an index value keeps track of which pixel column this corresponds to in the *CView*
- move to the center of the *CView* in the y-dimension
- draw a line DOWN to the circular buffer's audio sample -value/2
- move back to the center in the y-dimension
- draw a line UP to the circular buffer's audio sample value/2

You can see that instead of drawing a single vertical line segment for each cell, I am drawing two lines, one going up and one going down. There are two reasons I am choosing this logic:

1. the wave form will always be centered in the y-dimension
2. when using semi-transparent colors (as we do here) the second line segment that is drawn will share a common pixel (the center value) with the first line segment, forming a darker line that resembles an x-axis laying under the graph

OK, here is the complete drawing code. You might want to step through this code a few times to watch how the drawing occurs. I am leaving this in the normal document font to make the long lines fit. The drawing code is in **bold**.

```
void CWaveFormView::draw(CDrawContext* pContext)
{
	// --- bitmap, if one
	if(getDrawBackground())
	{
		getDrawBackground()->draw(pContext, size);
```

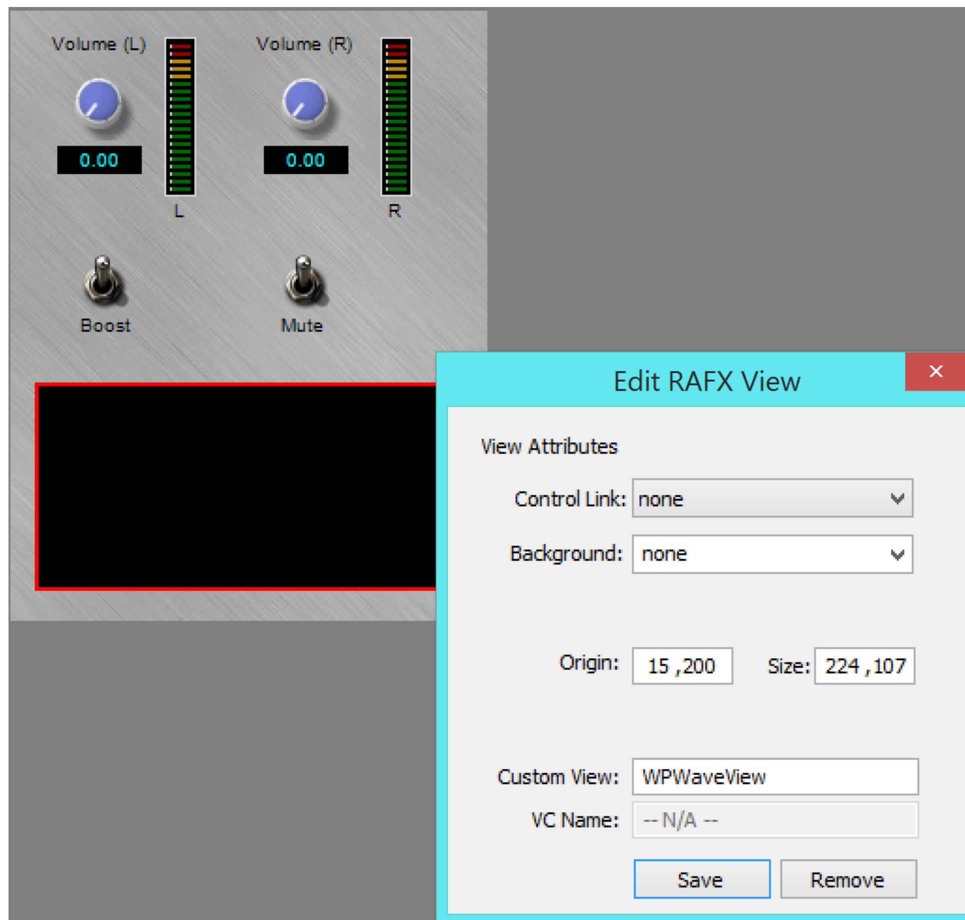
Copyright © 2015 Will Pirkle

```
	}
	else
	{
		// --- setup the background rectangle
		pContext->setLineWidth(1);
		pContext->setFillColor(CColor(200, 200, 200, 255)); // light grey
		pContext->setFrameColor(CColor(0, 0, 0, 255)); // black

		// --- draw the rect filled (with grey) and stroked (line around rectangle)
		pContext->drawRect(size, kDrawFilledAndStroked);

		// --- this will be the line color when drawing lines
		//     alpha value is 200, so color is semi-transparent
		pContext->setFrameColor(CColor(0, 0, 255, 200));

		if(!m_pCircBuffer) return;

		// --- step through buffer
		int index = m_nWriteIndex - 1;
		for(int i=1; i<m_nLength; i++)
		{
			float sample = m_pCircBuffer[index--];
			float normalized = sample*size.height();
			if(normalized > size.height() - 2)
				normalized = size.height();

			// --- halves
			normalized /= 2.0;

			// --- find the three points of interest
			const CPoint p1(size.left + i, size.bottom - size.height()/2.0);
			const CPoint p2(size.left + i, size.bottom - size.height()/2.0 - normalized);
			const CPoint p3(size.left + i, size.bottom - size.height()/2.0 + normalized);

			// --- draw lines
			pContext->drawLine(p1, p2);   // line from center to p2
			pContext->drawLine(p1, p3);   // line from center to p3

			// --- wrap the index value if needed
			if(index < 0)
				index = m_nLength - 1;
		}
	}
	setDirty (false);
}
```

With the waveform view code complete, we only need to hook it into the plugin. This waveform object will need to persist from one *showGUI()* call to the next so we need to make it a member variable. In addition, we will need a variable that contains the absolute value of the current audio output sample at any time. The VSTGUI timer will fire every 50 milliseconds so we will use the passing value of the output at that time for the plot. The sample code uses the left channel as the data source. The timer will NOT fire when there is no GUI.

Copyright © 2015 Will Pirkle

Here is where to set the Custom View name in the GUI Designer — it is named *WPWave-View*:



CustomViewsFour.h

In the plugin interface file, we need to #include the new waveform object (it is already added to the Visual Studio project) and create its variables for the waveform view and current audio output level:

```
#include "plugin.h"
#include "GUIViewAttributes.h"
#include "WaveFormView.h"
#include "../vstgui4/vstgui/vstgui.h" // for CView definition

class CCustomViews : public CPlugIn
{
public:

      <SNIP SNIP SNIP>

      // Add your code here: ------------------------------------------- //
      //
```

Copyright © 2015 Will Pirkle

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

    // --- custom waveform view
    CWaveFormView* m_pWaveFormView;

    // --- current output sample value
    float m_fCurrentOutput;

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

CustomViewsFour.cpp
Step through the required functions:

Constructor:
- clear the *m_pWaveForm* pointer
- clear the *m_fCurrentOutput* to 0.0

```
CCustomViewsFour::CCustomViewsFour()
{
    // 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_pWaveFormView = NULL;
    m_fCurrentOutput = 0.0;
}
```

prepareForPlay():
- clear the *m_fCurrentOutput* to 0.0

```
bool __stdcall CCustomViewsFour::prepareForPlay()
{
    // Add your code here:
    m_fCurrentOutput = 0.0;

    return true;
}
```

processAudioFrame():
- save the current left output value after all processing has been done

```
bool __stdcall CCustomViewsFour::processAudioFrame(float* pInputBuffer,
                                                    float* pOutputBuffer,
                                                    UINT uNumInputChannels,
                                                    UINT uNumOutputChannels)
{
    // --- volume
```

Copyright © 2015 Will Pirkle

```
	float fVolLeft = pow(10.0, m_fVolumeLeft_dB/20.0);
	float fVolRight = pow(10.0, m_fVolumeRight_dB/20.0);

	<SNIP SNIP SNIP>

	// --- do it
	pOutputBuffer[0] = pInputBuffer[0]*fVolLeft;
	m_fMeterValueL = fabs(pOutputBuffer[0]);

	// Mono-In, Stereo-Out (AUX Effect)
	if(uNumInputChannels == 1 && uNumOutputChannels == 2)
		pOutputBuffer[1] = pInputBuffer[0]*fVolLeft;

	// Stereo-In, Stereo-Out (INSERT Effect)
	if(uNumInputChannels == 2 && uNumOutputChannels == 2)
	{
		pOutputBuffer[1] = pInputBuffer[1]*fVolRight;
		m_fMeterValueR = fabs(pOutputBuffer[1]);
	}

	//  save left output for wave view
	m_fCurrentOutput = fabs(pOutputBuffer[0]);

	return true;
}
```

showGUI():
Decode the incoming message.

For the Custom View message we need to instantiate our object:

```
// --- module 4 ONLY!! This is our scrolling wave view
if(info->customViewName.compare("WPWaveView") == 0)
{
	// --- get the needed attributes with the helper
	const CRect rect = m_GUIHelper.getRectWithVSTGUIRECT(
						info->customViewRect);

	// --- create it!
	m_pWaveFormView = new CWaveFormView(rect);

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

For the GUI Timer Ping message we need to add the current output amplitude to the circular buffer and then call the *invalid()* function to force the control to repaint.

```
case GUI_TIMER_PING:
{
	if(m_pWaveFormView)
	{
		m_pWaveFormView->addWaveDataPoint(m_fCurrentOutput);
		m_pWaveFormView->invalid();
	}
	return NULL;
```

Copyright © 2015 Will Pirkle

```
}
```

Finally, for the GUI_WILL_CLOSE method, don't forget to NULL out the pointer so we don't use it accidentally:

```
case GUI_WILL_CLOSE:
{
	if(m_pWaveFormView)
		m_pWaveFormView = NULL;

	return NULL;
}
```

That's it! Compile the code and watch the audio scroll across the *CView*.

Object Destruction?
One final comment - you may notice that there is no code here to delete the *CWaveFormView* object when the GUI is closed. And, in the last module, there was no code to delete all the custom view objects we created. The reason is that VSTGUI4 uses reference counting to delete our objects when they are no longer needed.

Lastly, you also do not have to worry about the VST Timer Ping - it will only occur after the GUI has been created (or re-created after destruction). But we still check the validity of the pointer before using it.

In the next module, we will design a pure-custom GUI programmatically and without the RackAFX GUI Designer.

References:

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