



Anatomy of a custom Dplug widget

How do widget work?
Which callbacks to answer?
When to call setDirtyWhole? etc.



Meeting
Mar 8th 2022



Plug-in UIs are important

- It highlights the unique features of your plug-in.
If the UI doesn't say loudly that a feature is interesting, then it will be considered not-new.
- Custom widgets brings your UX desire to reality.
"I have a dream. I want to set both Ratio and Threshold at the same time"
- Dplug allows Scriptable + Resizable + PBR UI

2022 limitations: not DPI aware on Mac, no dynamic widget creation.

Let's build that widget : a stereo width control



It takes 183 LOC.

When dragged



When mouse is over



Rest position



Let's build that widget : a stereo width control

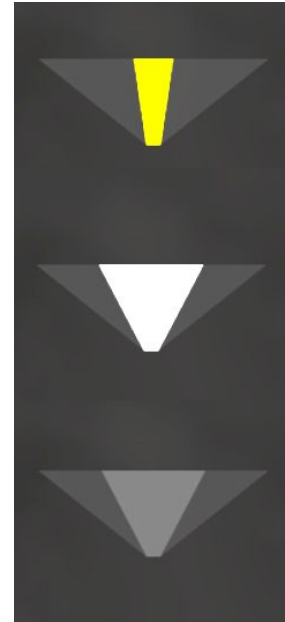
NOT THAT TRIVIAL TO CODE.

When dragged

It takes 183 LOC.

When mouse is over

Rest position





Let's go over those 183 lines, one by one.

THE HEADER

personal canvas
helpers functions

simple UI vs DSP
communication

easiest way to
draw widget

won't be
derived

root class of all UI widgets

```
module widthcontrol;


import gui;
import core.atomic;
import dplug.core;
import dplug.gui;
import dplug.canvas;
import dplug.client;

/// A stereo width control,
/// takes one parameter that goes 0 to 200%
final class UIWidthControl : UIElement, IParameterListener
{
public:
nothrow:
@nogc:
```

receives
parameter
changes

You should need to import that class name from Wren. The name don't `_have_` to start with UI, I like to do that for UIElement derivatives.

THE SCRIPTED PROPERTIES



```
@ScriptProperty RGBA colorOn = RGBA(90, 90, 90, 200);
@ScriptProperty RGBA colorOff = RGBA(65, 65, 65, 200);
@ScriptProperty RGBA colorDragged = RGBA(255, 255, 0, 255);
@ScriptProperty RGBA colorHovered = RGBA(255, 255, 255, 255);
@ScriptProperty RGBA colorDisabled = RGBA(65, 65, 65, 200);
@ScriptProperty float sensivity = 0.06f;
```

Expose what you want for Wren to see.

Tip: if you draw something complicated, use temporary @ScriptProperty that are not “styling” but content. (eg: control points in a curve)

THE CONSTRUCTOR & DESTRUCTOR

Flags (see next slides)

Stereo width parameter

The “section enable” BoolParameter

Register and unregister the widget as a Listener for each parameter that will change the graphics when changing.

DO NOT FORGET TO BALANCE CALLS:

- `addListener`
- `removeListener`

```
this(UIContext context, Parameter param, Parameter enableParam)
{
    super(context, flagRow);
    _param = cast(FloatParameter) param;
    _param.addListener(this);

    _enableParam = cast(BoolParameter) enableParam;
    _enableParam.addListener(this);

    clearCrosspoints();
}

~this()
{
    _enableParam.removeListener(this);
    _param.removeListener(this);
}
```




flagRaw ⇔ **onDrawRaw** is called.

This widget can write stuff on the Raw level, always on top of the PBR level. Suitable for 60FPS.

flagAnimated ⇔ **onAnimate** is called

Typically used to request a redraw if things have changed.

flagPBR ⇔ **onDrawPBR** is called

This widget can write stuff on the PBR level.

*This is **not** suitable for 60FPS display, so this sort of widget has to call `setDirty` infrequently.*

DRAWING PART 1



Raw texture, cropped to the widget position. Pixel (0,0) is top-left of **widget**.

You can modify **only** inside the `dirtyRects`. Their position are relative to `rawMap` (local coordinates).

Get widget size by reading `position()`. This is the “world” position, not the position in `rawMap`.
Only worthwhile for width and height.

Repeat cropped drawing for each `dirtyRect` since it's generally 1 or 2 rects.

```
override void onDrawRaw(ImageRef!RGBA rawMap, box2i[] dirtyRects)
{
    bool enabled = _enableParam.value();

    float W = position.width;
    float H = position.height;

    float center = W * 0.5f;
    float extent = W * 0.49f * 0.8f; // more esthetic to leave a bit of secondary color
    float baseExtent = W * 0.03f; // so that a line does appear for 0% width

    float width = _param.getNormalized(); // 0 to 1

    foreach(dirtyRect; dirtyRects)
    {
        auto cRaw = rawMap.cropImageRef(dirtyRect);
        canvas.initialize(cRaw);
        canvas.translate(-dirtyRect.min.x, -dirtyRect.min.y);
    }
}
```

`translate()` call => so that you can draw in local coordinates.

Pro-tip = Have key metrics in the widget depend on the widget size (width and height).

This allows to:

1. Have less work to do in `reflow()` (eg: `fontSizePx`)
2. Resize things interactively with right-click in debug mode.
3. Easier to scale UI.

```
override void onDrawRaw(ImageRef!RGBA rawMap, box2i[] dirtyRects)
{
    bool enabled = _enableParam.value();

    float W = position.width;
    float H = position.height;

    float center = W * 0.5f;
    float extent = W * 0.49f * 0.8f; // more esthetic to leave a bit of secondary color
    float baseExtent = W * 0.03f; // so that a line does appear for 0% width

    float width = _param.getNormalized(); // 0 to 1

    foreach(dirtyRect; dirtyRects)
    {
        auto cRaw = rawMap.cropImageRef(dirtyRect);
        canvas.initialize(cRaw);
        canvas.translate(-dirtyRect.min.x, -dirtyRect.min.y);
    }
}
```

>>> AVOID PIXEL QUANTITIES IF YOU CAN! <<<

DRAWING PART 2

```
foreach(dirtyRect; dirtyRects)
{
    auto cRaw = rawMap.cropImageRef(dirtyRect);
    canvas.initialize(cRaw);
    canvas.translate(-dirtyRect.min.x, -dirtyRect.min.y);


    // Fill with off color
    canvas.fillStyle = colorOff;
    canvas.beginPath();
    canvas.moveTo(0, 0);
    canvas.lineTo(W, 0);
    canvas.lineTo(center + baseExtent, H);
    canvas.lineTo(center - baseExtent, H);
    canvas.closePath();
    canvas.fill();

    // Fill with on color
    RGBA color = colorOn;
    if (isMouseOver) color = colorHovered;
    if (isDragged) color = colorDragged;
    if (!enabled) color = colorDisabled;

    canvas.fillStyle = color;
    canvas.beginPath();
    canvas.moveTo(center - extent*width - baseExtent, 0);
    canvas.lineTo(center + extent*width + baseExtent, 0);
    canvas.lineTo(center + baseExtent, H);
    canvas.lineTo(center - baseExtent, H);
    canvas.closePath();
    canvas.fill();
}
```

The drawing itself.

Using `@ScriptProperty` values instead of hardcoded values will make the widget more reusable.



Important: A widget is **not** redrawn unless you call `setDirtyWhole` .

Called when the parameter is changed by UI interaction **OR** host automation.

=> call `setDirtyWhole` since DAW automation wouldn't redraw else.

Called when the UI calls `beginParamEdit` / `endParamEdit()` .

```
88
89     override void onParameterChanged(Parameter sender)
90     {
91         setDirtyWhole();
92     }
93
94     override void onBeginParameterEdit(Parameter sender)
95     {
96         setDirtyWhole();
97     }
98
99     override void onEndParameterEdit(Parameter sender)
100    {
101        setDirtyWhole();
102    }
103
```

=> we call `setDirtyWhole` there too, unless you do it when you start dragging, or stop dragging. Perhaps not necessary in that widget. Most often this can be left empty because you will have `onMouseClicked` / `onBeginDrag` / `onStopDrag`...

SETDIRTY AND SETDIRTYWHOLE



You can optimize rendering of a large widget by redrawing only the portion of its position rectangle, that has changed.

setDirty

VS

setDirtyWhole ()

Redraws a single rectangle area, given in local coordinates. And the widgets beneath it.

Redraws the whole position rectangle, and the widgets beneath it.

Tip: Partial rectangles are generated anyway when another window pass above your plug-in, so you have to handle it anyway. And that's why `dirtyRects` exists.

Parameter reset code. Can't set parameters outside of a **balanced beginParamEdit/endParamEdit** pair.

```
override bool onMouseClick(int x, int y, int button, bool isDoubleClick, MouseState mstate)
{
    // double-click => set to default
    if (isDoubleClick || mstate.altPressed)
    {
        _param.beginParamEdit();
        _param.setFromGUI(_param.defaultValue());
        _param.endParamEdit();
    }
    return true; // to initiate dragging
}
```

VERY IMPORTANT


Returning **true** = **Start a drag operation**. Window captures mouse until release.

If you don't need to do anything while mouse dragging, you still have to return true to consider the click handled.

Returning **false** = Consider the click unhandled. *The event passed down to children etc.*

RESPONDING TO A MOUSE DRAG Part 1

A mouse drag happens whenever `onMouseClicked` returned `true`.



Dragging has a start and an ending,
it can be used to call balanced pairs
of `beginParamEdit()` /
`endParamEdit()`

MAKE SURE THOSE CALLS ARE

BALANCED

WHATEVER HAPPENS,
AND IN WHICHEVER CALLBACK

```
override void onBeginDrag()  
{  
    _param.beginParamEdit();  
}  
  
override void onStopDrag()  
{  
    _param.endParamEdit();  
}
```

Tip: eventually use a state machine to know which parameter you are dragging, it is useful for controls that have several points to drag.

RESPONDING TO A MOUSE DRAG Part 2

Use mouse displacement dx and dy
(not as precise to use x and y, but can be done)

```
override void onMouseDrag(int x, int y, int dx, int dy, MouseState mstate)
{
    // FUTURE: replace by actual trail height instead of total height
    float displacementInHeight = cast(float)(dy) / _position.height;

    float modifier = 1.0f;
    if (mstate.shiftPressed || mstate.ctrlPressed) modifier *= 0.1f;

    double oldParamValue = _param.getNormalized();
    double newParamValue = oldParamValue - displacementInHeight * modifier * sensitivity;
    if (mstate.altPressed)
        newParamValue = _param.getNormalizedDefault();

    if (y > _mousePosOnLast0Cross) return;
    if (y < _mousePosOnLast1Cross) return;
    if (newParamValue <= 0 && oldParamValue > 0) _mousePosOnLast0Cross = y;
    if (newParamValue >= 1 && oldParamValue < 1) _mousePosOnLast1Cross = y;
    if (newParamValue < 0) newParamValue = 0;
    if (newParamValue > 1) newParamValue = 1;
    if (newParamValue > 0) _mousePosOnLast0Cross = float.infinity;
    if (newParamValue < 1) _mousePosOnLast1Cross = -float.infinity;

    if (newParamValue != oldParamValue)
    {
        if (auto p = cast(FloatParameter)_param)
        {
            p.setFromGUINormalized(newParamValue);
        }
        else
            assert(false); // only float parameters supported
    }
}
```


Using dx/dy
needs to read
current
parameter
value.

Slider logic
like UISlider

Divide by height
to be as
sensitive at
every plugin
size.

SHIFT+ clic
= finetune

WHO IS CALLED?
within dragging:
=> **onMouseDown**
without dragging
=> **onMouseMove**



Called when mouse enters the widget. Here we trigger redraw because it may trigger selection highlight.

```
override void onMouseEnter()  
{  
    setDirtyWhole();  
}  
  
override void onMouseExit()  
{  
    setDirtyWhole();  
}
```

Also called here because it may lose selection highlight.

onMouseMove may not be generated when the mouse exits the widget!

=> Use **onMouseExit** if you have a visual change just by hovering the mouse.

Must hold its own Canvas, because several widget can be drawn simultaneously if they don't intersect. **Don't share Canvas instances.**

Hold parameters
it reads/write values from

Slider logic taken
from UISlider

```
169
170 private:
171     Canvas canvas;
172     FloatParameter _param;
173     BoolParameter _enableParam;
174
175     float _mousePosOnLast0Cross;
176     float _mousePosOnLast1Cross;
177
178     void clearCrosspoints()
179     {
180         _mousePosOnLast0Cross = float.infinity;
181         _mousePosOnLast1Cross = -float.infinity;
182     }
183 }
```



FUTURE

- Documenting all that a bit more.
- 3 possible return values for **onMouseClicked**
 - Event consumed, start Dragging.
 - Event consumed, do not start Dragging.
 - Event not consumed.



Questions?