UI Feedback optimizations in Lens

How to have efficient, large-scale feedback in a Dplug plug-in.
Lens plug-in = more feedback than typical for us

GAIN MAP
Display energy estimate, and gain reduction for compressor and expander in gain map.

EQ
Each EQ has spectrogram, one of those display also gain reduction from compressor.
What are the desirable properties of DSP to UI feedback?

- **Fast.** It should be cheap enough to be activated at all time. *(Thankfully dplug:canvas is fast)*
- **Large.** People want many things to be feedback visually.
- **Non blocking.** It’s not worth holding back the audio if the UI struggles.
- **Decoupled from DSP buffer size.**
- **Decoupled from DSP sampling rate.**
- **Sync.** Visual should approximately correspond to audio temporally.
Why not just use \texttt{core.atomic}?

```
shared(int) V;
int V_display;

onAnimate "V changed, call setDirty etc set V_display"

onDrawRaw
draw with V_display
```

process "send measured value of V"

```d
atomicStore
onAnimate "V changed, call setDirty etc set V_display"
```
Why not just use `core.atomic`?

- Fast. ✅
- Large. ❌ (it's just one scalar)
- Non blocking. ✅
- Decoupled from DSP buffer size. ❌
- Decoupled from DSP sampling rate. ❌
- Sync. ❌
Why not just use a mutex?

- Fast. ✅
- Large. ✅
- Non blocking. ❌
- Decoupled from DSP buffer size. ❌
- Decoupled from DSP sampling rate. ❌
- Sync. ❌

```cpp
#include <iostream>

int V;
int V_display;

lock/unlock

// DSP
process "send measured value of V"

// main.d
int V;  
int V_display;

lock/unlock

// UI
onAnimate "V changed, call setDirty et set V_display"

onDrawRaw
draw with V_display
```

- Fast. ✅
- Large. ✅
- Non blocking. ❌
- Decoupled from DSP buffer size. ❌
- Decoupled from DSP sampling rate. ❌
- Sync. ❌
What’s the problem with (large) buffer size?

Say we have 400ms buffer size, and the plugin is 20x realtime. Takes 20ms to process audio.

- **Audio output**: 400ms of audio (buffer size)  
- **DSP thread**: DSP (20ms)  
- **UI thread**: Draw Feedback  
  
Whole UI state for 400ms of audio events here
What’s the problem with (large) buffer size?

Say we have 400ms buffer size, and the plugin is 20x realtime. Takes 20ms to process audio.

- Audio output: 400ms of audio (buffer size)
- DSP thread: DSP (20ms)
- UI thread: Draw Feedback

Whole UI state for 400ms of audio events here

FEEDBACK IS FROZEN HERE
What’s the problem with (large) buffer size?

Say we have 400ms buffer size, and the plugin is 20x realtime. Takes 20ms to process audio.

400ms of audio

DSP thread

DSP (20ms)

UI thread

Draw Feedback

FEEDBACK FROM BACK OF BUFFER IS DISPLAYED BEFORE THE AUDIO HAPPENS

Whole UI state for 400ms of audio events here

400ms of audio

Draw Feedback

FEEDBACK IS FROZEN HERE

Feedback from back of buffer is displayed before the audio happens.
You need a queue somewhere.
You need a queue somewhere.

`onAnimate` should read the queue progressively, and *at the right speed*. 
That’s where TimedFIFO originated (2016).
Using `dplug.core.ringbuf.TimedFIFO` for managing audio buffer sizes and processing in DSP and UI threads.

**Audio output**
- Enqueue 400ms of state in TimedFIFO + samplerate
- Dequeue and draw feedback progressively, samplerate = in-band

**DSP thread**
- DSP (20ms)
  - Enqueue 400ms of state in TimedFIFO + samplerate

**UI thread**
- DSP (20ms)
TimedFIFO

- Fast. ✅
- Large. ✅
- Non blocking => almost
- Decoupled from buffer size => almost
- Decoupled from sampling rate => ...no! ❌
- Sync. ✅

```
DSP

trylock/unlock

process "send measured value of V with audio samplerate"

main.d

trylock/unlock

TimedFIFO!int Vqueue;
int[N] V_display;

onAnimate "use the queue, call setDirty if anything changed"

gui.d

trylock/unlock

onDrawRaw
draw N last samples with V_display
```
TimedFIFO problems

- **Samplerate.** Optimal queue size depends on sample rate, which changes everything again. At 96000 Khz, queue is emptied faster. But no knowledge of sampling rate at queue startup.

- **(typically) Buffer size.** Even if you gather feedback every 32 samples, if your buffer size if not multiple of 32, you will forget some.

- **Non-blocking.** One trylock is kinda ok for the whole feedback, but the problem is that typical plugin have several such TimedFIFO.

  This is where we stopped for earlier Auburn Sounds plugins.
LENS compressor = more feedback than typical for us

Compressor Input volume (64 bands)
Expander Input volume (64 bands)
Compressor Gain reduction (64-bands)
Expander Gain reduction (64-bands)
Spectral volume for sidechain (64-bands)
Spectral volume for wet signal (64-bands)
etc...

The whole unit of feedback: 519 scalar values transit from DSP to UI, measured 40x per second.
Feedback Tip #1: single FeedbackData struct

1. Have one single FeedbackData struct for all plugin feedback.

2. Have one single TimedFIFO for that

3. Profit from less synchronization

```c
struct FeedbackData
{
    float sampleRate;
    int numBins;
    float minRateHz;
    float maxRateHz;
    bool listenMode;  // Listen to sidechain
    bool relativeMode; // Relative mode

    // So that we don't need to dirty things if the generation is inferior.
    long generation;

    // ***************** OTHER FEEDBACK ARRAYS ******************/
    float inputExpand linear; // Volume of expander input, possibly normalized (relative).
    float outputExpandGR linear;
}```
Feedback Tip #2: compute feedback 40x/sec, and for a single sample.
in `processAudio` callback

```c
// Should we collect feedback in this callback?
// and in which sample?
// If collectFeedback is true, this will be recomputed.
bool collectFeedback = false;
int feedbackSamplePos = 0;
if (_feedbackCounter == -1) // initialization
{
    _feedbackCounter = 0;
    collectFeedback = true;
}

_feedbackCounter += frames;
if (_feedbackCounter >= _collectFeedbackEverySamples)
{
    collectFeedback = true;
    _feedbackCounter = _feedbackCounter % _collectFeedbackEverySamples;
    feedbackSamplePos = frames - 1 - _feedbackCounter;
    assert(_feedbackCounter >= 0 && _feedbackCounter < frames);
    assert(feedbackSamplePos >= 0 && feedbackSamplePos < frames);
}

// with _collectFeedbackEverySamples = cast(int)(sampleRate / FEEDBACK_DSP_HZ + 0.5f)
```
in `processAudio` callback

```cpp
// Should we collect feedback in this callback?
// and in which sample?
// If collectFeedback is true, this will be recomputed.
bool collectFeedback = false;

if (collectFeedback == true) {
  // ONLY COMPUTE THE FEEDBACK STRUCT IF
  // collectFeedback == true

  AND THEN, ONLY FOR ONE SAMPLE
  IN THE WHOLE SUB-BUFFER.

  PASS THAT INFO IN ALL DSP THAT HAS
  FEEDBACK.

  _feedbackCollector.collectFeedback();
  _feedbackCollector.feedbackCollection();
  asserted
  asserted
}

// with _collectFeedbackEverySamples = cast(int)(sampleRate / FEEDBACK_DSP_HZ + 0.5f)
```
In lens, max subbuffer size is 512 samples, and feedback period is 1102 samples at 44100Hz. Feedback will only break down at 11025Hz.
Instead of pushing the audio samplerate in-band, give the FIFO the *feedback sampling rate*. (here = 40Hz) FIFO created with 12 slots, corresponding to 12 * 1000/40 ms of feedback independently of the audio sampling rate. 🎉
Annoying, but worth it.

- Fast.
- Large.
- Non blocking: *almost*
- Decoupled from buffer size.
- Decoupled from sampling rate.
- Sync.
Feedback Tip #3: accumulate delta time when `onAnimate` is called with small `dt`

```java
override void onAnimate(double dt, double time)
{
    _rateLimitDt += dt;
    if (_rateLimitDt > minimumAnimationDelta)
    {
        bool dirty = rateLimitAnimation(_rateLimitDt);
        if (dirty)
        {
            setDirtyWhole();
            _rateLimitDt = 0;
        }
    }
}
```
Feedback Tip #3: accumulate delta time when `onAnimate` is called with small `dt`.

```java
override void animate(double dt, double time) {
    _rateLimitDt += dt;
    if (_rateLimitDt > minimumAnimationDeltaD) {
        bool dirty = rateLimitAnimation(_rateLimitDt);
        if (dirty)
            setDirtyWhole();
        _rateLimitDt = 0;
    }
}
```

100 ms

Save CPU by avoiding some redraw.

Basically = not worth it to redraw for too small a change.
Feedback Tip #4: Fix your timestep when needed.

- `onAnimate` is called repeatedly, but with any variable delta time ($dt$).

- Like in video games, this can be tricky for animation, especially if you want points with trails.

- But you can manually fix your timestep for some widgets.

```java
override void onAnimate(double dt, double time) {
    // Implement your logic here
}
```
Fixed animation rate howto

```java
override void onAnimate(double dt, double time) {
    // Sub animation, with fixed frame-rate.
    _accumulatedDt += dt;

    float decayAlpha = 1.0 - expDecayFactor(rms Decay Time, 1.0 / animationStep);

    while(_accumulatedDt > animationStep) {
        _accumulatedDt -= animationStep;
        if (animationFrame(decayAlpha))
            dirty = true;
    }

    if (dirty == true)
        setDirtyWhole();
}
```

100ms eventuall hoist computation out of the fixed animation toop (unsure gain here)

animationFrame called 10x per second
Feedback Tip #5: Drawing performance.

Same old advice.

- **Use dplug:canvas**, it write 4 pixels at once.

- **Do not update PBR layer for animation**, except for small widgets.

- *(advanced)* You can dirty only the graphics subpart of the widget that you know will be affected.

- Things will draw faster if update area rectangle is small and constrained. But, hard to do.
Questions?

Thanks for listening!