universe vst

synthesizer by xoxos

Universe is inspired by cepstral resynthesis, which separates "source" signals from "resonances" in the spectral domain. This process breaks spectral data into four arrays: magnitude, frequency, and phase of the source, and magnitude of resonances. Instead of deriving these four arrays from sample analysis, Universe populates them using contours.

The limitation of my development environment prohibits graphic editing of thousands of partials. My solution was to create a global contour array which is referenced by an index point, and translated to data with scale, log (how scale is stretched across the spectrum), and amount.

The easiest way to understand this feature is to load the default preset and view the spectrum in a spectrum analyser. You will initially see a flat series of partials. Turn up the MAG AMT knob at the top of the gui, and you will see a contour applied to the spectrum. Adjusting FINE, SCALE and LOG will move the spectral features. The leftmost INDEX parameter moves the reference point at a larger scale.

The contour array was created with cubic interpolated noise and contains a variety of adjacent features. Attempting to locate and match these features to your intention isn't the most graceful way to patch a timbre but it allows complex shaping to be done in four parameters, and allows some unique modulations.

When you play a different note on the keyboard, the contour will be retained by the partials, like a standard synthesizer oscillator. The CONV contour is applied to the spectrum afterwards, without tracking the pitch. It is much easier to observe this contour when the MAG AMT is flat. The idea is that some approximation of real world acoustics can be approached. The CONV contour only reduces the volume of partials, so higher amplification is necessary.

It is harder to observe the FREQ contour, which moves partials up or down in pitch in relation to the fundamental. An option titled F0 allows the fundamental to retain its original pitch or not, though generally the partials affect perception of pitch as well.

The PHAS contour controls phasiness of each partial by adding a random amount each frame. The unique RATE knob lowpasses these values to create smooth phase variance or noisy.

The PHAS component of the synthesizer allows for efficient unison voicing by simply keeping track of extra phase values. The limit of this method is that the pitch variance of each partial remains in its "bin" so that longer frames and higher pitches have perceptually less detune without adding the extra computations to see which bin the voice would belong in otherwise. Maybe next time. Reducing the window to 1024 or 512 samples is sufficiently phasy for typical unison pads.

There is an option to opt for only odd harmonics if you like those.

the technical bits

Understanding the process used to render the signal is necessary for practical use. As you may know, fourier processes (FFT) audio in blocks, here defined in length by the WINDOW parameter.

The quality of rendering is increased with more overlapping windows. Increasing overlaps reduces the volume of sidebands to rendered partials, which can be observed in a spectral analyser by simplifying the patch.

Changing the window length or the number of overlaps changes the frequency at which the partials must be computed, essentially doubling or halving it with every increment. Around 1024 or 2048 is usually the most efficient. The problem with longer window rates is that pitch changes produce a diffuse signal since each frame is rendered at a fixed pitch. Low window rates are much better but might be excessively intensive to compute. This architecture is more suited for diffuse, static pitched timbres than portamento.

the problem

Universe uses a constant frame rate to render the signal. It is also possible to build a phase vocoder architecture that performs pitch changes by reading the rendered file at variable speeds, which results in lower cpu with lower pitch and higher cpu with higher pitch. I used the constant cpu method here.

The problem is that when a new note is played at a new pitch, Universe moves to the next frame to render. The previous frames, which are still heard in the overlap, contain information written at the old pitch. So when a new note is played, it contains a fragment of audio at the previous pitch played in the voice.

The reason this is a problem is because choosing to wipe the buffer or leave the old frames depends on whether we are picking up on a voice legato, or the voice has ended, but there is no way to differentiate these events in the SDK. Which is alright, it's been many years :)

Using a shorter window time will shorten the time of the artifact. Patches with long attacks will not suffer if the previous voice has ended, neither will patches playing the same pitch. If you can't use a separate instance for each pitch or put up with the artifact, then get me an egalitarian society I can publish for in the contemporary mode.

Perhaps the other problem is the "stereo" signal, which is produced by a single FFT process so that the right channel is 90 degrees out of phase from the left (if you didn't know that about FFT piggybacking). It's good to be aware of that sort of thing when you are building sound.

the solution

Universe took a couple of weekends to make, I'd never done an FFT rendered oscillator. The poignant thing here is that using fourier processing to render is amazingly simple to do. Once I had coded the basic script which rendered sawtooth partials at variable pitch and constant frame rate, I realised the entire script for the algorithm was around thirty lines. Any kind of synthesizer could be created by specifying the partials, if the public are empowered.

The method is super simple to understand. If you have 44100 samples per second and use a 2048 window length, a new window begins every .0464399 seconds. If you use 4x overlap, this rate drops by four. This is the hop size, or time between frames. Once you have this time, you use the pitch of the partial you wish to render to determine how many cycles it has advanced in that time, by simple multiplication. The decimal part of this result is the phase advancement, since each whole number is an entire cycle.

Here's the process loop... flip is used to track when its time to call the frame renderer, then t is used to sum the audio from the overlapping frames

```
register float t;

flip = p * overlap;	flip >>= window;
if (flipbuf != flip) {
//	float *in2 = buffer_offset + input2_buffer;
	float hz = pow(2.f, *in2 * 10.f) * 13.75f;
	if (winbuf != window || overbuf != overlap) {
		n = powtwo[window];nm1 = n - 1;	nd2 = n >> 1;
		hn = n / overlap;	//	hop size in samples
		winbuf = window;	overbuf = overlap;
		for (int i = 0; i < n; i++) {
			hann[i] = (1.f - cos(tau * (float)i / (float)n)) / (float)overlap;
	}	}
	hr = hz * (float)hn / samplerate;	//	hr = hop size in seconds but its times hz also for efficiency
	bin = hz / (samplerate / (float)n);
	rend(wl[flip], wr[flip]);
}
flipbuf = flip;


t = wl[0][p];
for (int i = 1; i < overlap; i++) {
	int pt = p - i * hn;	while (pt < 0.f) pt += n;
	t += wl[i][pt];
}
*out1 = *out2 = t;

p++;
if (p >= n) p = 0;
```

Here's the render routine. The first part sets the magnitude and frequency ratio of a sawtooth. The source used for Universe is slightly more complicated to add in the phase randomness and some efficiency redundancies, but seriously.. this is a working script that should be seen as accessible.

```
float* mags;    mags = (float*) malloc (sizeof(float) * n);
float* phas;    phas = (float*) malloc (sizeof(float) * n);

for (int i = 0; i < partials; i++) {        //      temporary construct partial attributes
        pmag[i] = 1.f / (float)(i + 1);
        pfr[i] = i + 1;
}

for (int i = 0; i < partials; i++) {        //      partials to bin array
        float bd = bin * pfr[i];
        int bix = (int)bd;      bd -= (float)bix;
        if (bd >= .5f) {
                bd -= 1.f;      bix++;}
        if (bix < nd2) {
                mags[bix] += pmag[i] * cos(ppha[i]);
                phas[bix] += pmag[i] * sin(ppha[i]);
        }

        bd = pfr[i] * hr;       //      reuse bd
        bd -= floor(bd);
        ppha[i] = unwrap(ppha[i] - bd * tau);
}

fft(window, mags, phas);

for (int i = 0; i < n; i++) {
        real[i] = mags[i] * hann[i];
        imag[i] = phas[i] * hann[i];
}
```

In case phase unwrapping is a mysterious concept because you haven't done phase vocoder.. just stripping the phase to the relevant data.

```
inline float unwrap(float phas) {float period = phas * 0.159154943f;        return phas - floor(period + .5f) * tau;};
```

amazing that, apart from the bits of code to input parameters and stuff, and the fft routine, that's all the instruction required to build an IFFT based synth and do all kinds of nifty timbral things. I''m making this with the same environment as was available in 2002. This synth uses ~10% for one voice average patch... why aren't there hundreds of FFT based synths for free?