"Design of the THX Deep Note"

excerpt (pp. 176-181) from Artful Design, Chapter 4 "Programability and Sound Design"

https://artful.design/
The THX Deep Note!

To illustrate sound synthesis by way of parametric evolution, we are going to recreate one of the most recognizable pieces of computer-generated sound ever designed: The THX Deep Note!

Designed and programmed in 1982 by James Andy Moorer (also a founding member of CCIRMA), the Deep Note was first introduced with the 1983 Premier of Return of the Jedi and has been heard in countless THX trailers for movies and video games!

When Andy created the Deep Note, he was an employee of Lucasfilm’s Computer Division (which not only led to THX but eventually Pixar). THX creator Tom Holman asked Andy to create a sound logo that “comes out of nowhere and gets really, really big.”

In 1982, it took Andy Moorer 325 lines of C code running on a specialized hardware and software Audio Signal Processor. Here we are going to recreate it in ChucK! It won’t be exactly the same, but we will try to capture the essence of the sound design!

The Deep Note was synthesized using 30 voices with randomized starting frequencies between 40Hz to 350Hz. These voices smoothly glide toward a predetermined chord spanning 6 octaves, over a duration of 30 seconds.

A Plan...

Setup Stage: Create provisions for 30 voices. In our case, we will instantiate 30 sawtooth wave generators, randomizing their respective starting frequencies (our emulation will use 160-360Hz as the starting range). Each voice will eventually reach one of 9 predetermined target frequencies.

Initial Stage: Begin the sound by ramping up the voices in amplitude (while holding the starting frequencies constant). The original Deep Note does something more sophisticated -- we’ll only approximate it here. The goal is to create the part of the sound that “comes out of nowhere.”

Converging Stage: Gradually change the frequencies of all the voices toward their respective target frequencies. Accomplished by updating each voice’s frequency every so often (every 10-45s), so that it smoothly approaches the target (much like our zeno’s interpolator in Chapter 3, except this interpolation is linear). Here, the sound gets “really big”!

Target Stage: All voices reach their target frequencies at precisely the same time. Sounding our predetermined chord and creating an epic and unmistakable sense of arrival and resolution! We will hold this chord briefly before fading out.

We can illustrate the program graphically -- 30 lines represent the frequencies of the 30 voices over time. Observe the three stages the sound goes through!

Convergence

The voices smoothly glide toward their respective target frequencies. Overall, 30 voices converge on 9 target frequencies in a giant 30-way 

Chaos

30 frequencies randomized between 160-350Hz give an unsettling, brooding feeling -- “something is about to happen…”

The 9 Target Frequencies are just-intoned: the intervals between them are tuned as ratios of small integers. Mathematically, this “lines up” harmonics in the notes and sonically results in a big, stable, and pure sound.

Order + Resolution

The target frequencies stack up to a big chord spanning multiple octaves and giving a sense of epic resolution and arrival -- whoa.

Approximate Piano Keys for the Target Frequencies: Middle C for reference.
setup

// D1, D2, D3, D4, D5, A5, D6, F#6, A6
[ 37.5, 75, 150, 300, 600, 900, 1200, 1500, 1800,
  37.5, 75, 150, 300, 600, 900, 1200, 1500, 1800,
  37.5, 75, 150, 300, 600, 900, 1200, 1500, 1800,
  150, 300, 600, 900, 1200
] @=> float targets[];

float initials[30];
3.0::second => dur CHAOS_HOLD_TIME;
5.5::second => dur CONVERGENCE_TIME;
3.5::second => dur TARGET_HOLD_TIME;
2.0::second => dur DECAY_TIME;
SawOsc saw[30];
Gain gainL[30];
Gain gainR[30];
NRev reverbL => dac.left;
NRev reverbR => dac.right;
0.075 => reverbL.mix => reverbR.mix;
for( 0 => int i; i < 30; i++ )
{
  saw[i] => gainL[i] => reverbL;
  saw[i] => gainR[i] => reverbR;
  1.0 - gainL[i].gain() => gainR[i].gain;
  Math.random2f( 160, 360 ) => initials[i] => saw[i].freq;
}

for reference, this is our Deep Note Emulation Algorithm as a ChucK program, in four sections corresponding to our initial plan.

"chaos"

1.
now + CHAOS_HOLD_TIME => time end;
while( now < end )
{
  1 - (end-now) / CHAOS_HOLD_TIME => float progress;
  for( 0 => int i; i < 30; i++ )
  {
    0.1 * Math.pow(progress,3) => saw[i].gain;
  }
  10::ms => now;
}

"convergence"

2.
now + CONVERGENCE_TIME => end;
while( now < end )
{
  1 - (end-now)/CONVERGENCE_TIME => float progress;
  for( 0 => int i; i < 30; i++ )
  {
    initials[i] + (targets[i]-initials[i])*progress => saw[i].freq;
  }
  10::ms => now;
}

"resolution"

3.
TARGET_HOLD_TIME => now; // hold the chord!
now + DECAY_TIME => end;
while( now < end )
{
  (end-now) / DECAY_TIME => float progress;
  for( 0 => int i; i < 30; i++ )
  {
    0.1 * progress => saw[i].gain; // fade
  }
  10::ms => now;
}

Don’t worry if your eyes start watering from looking at this code — this is just to give a general idea of how we can use code to control sound over time.
BUILD COMPLEXITY AS THE SUM OF SIMPLE ELEMENTS

An audio-specific version of visual design principle 3.5: Build complexity from simplicity.

Computers are really good at making copies. Once we can program one thing, it’s trivial to instantiate more of it. The aim is not merely to have more, but to create something new in the amalgam.

For example, our deep note emulation is achieved through the addition of 30 basic sawtooth voices, modulating their frequencies in a specific and synchronized way. This produces the sense of a single, coherent sound! We might still hear individual voices in the mix, but we also hear the sum total of the voices as a culminating, cohesive sound.

The key here is not only that we have many voices, but that each one is both independently changing in frequency and globally coordinated with the other voices.

There are several programmability and design ideas in motion here, including precision of control, sonic narrative, and strength in numbers of simple elements acting together to culminate in a single pronounced effect.

Design things with a computer that would not be possible without!

Do not simply copy, port, digitize, or emulate. Rather, create something novel and unique to the medium -- something that could not exist without it.

It’s tempting to remake what already exists. While that remains a useful exercise, many people do that because it’s obvious. But with new technological mediums also come the opportunity and responsibility to discover what the medium is intrinsically good at.

This is an essential guiding principle of artful design (with any medium or technology). Let’s apply this lens and deconstruct a computer music composition -- one that uses the computer as a kind of personal musical filter to the world.

Two key components in creating complexity from simple elements

Local independence

Each element can change on its own.

Global coordination

All elements subject to a larger organizing principle.

Remember this from Chapter 3? One flare, multiplied by 500, arranged in a shimmering stream, where each flare twinkles and oscillates independently...