# Clothing Like A T-shirt Saying I Wish
# The Report

20060219

Nicolas Léveillé <nicolas.leveille@free.fr>

# Contents


I Introduction 1

1 Foreword 1

2 Description 2

II Design and Methodology 2

3 Where Am I 2

4 Where to go 3

5 How to do it 4

III Implementation 5

6 Rationale 5

7 Materials 5

8 How to build a synth in two days 6
- 8.1 Drums . . . . . . . . . . . . . . . . . . . 6
- 8.2 Strings . . . . . . . . . . . . . . . . . . . 6

9 Causes and consequences 7

10 Code reuse via Mixin-style programming 9

IV Conclusion 9

V Appendices 12

11 Drums 12

12 String phaser 13

13 FDN Reverb 14


# Part I
# Introduction

## 1 Foreword

In 2005, after a couple of years of ambitious, year-long objectives that never really manifested, I decided it was time to release again. I naturally became interested in finding out what really is the difference between fooling around and being productive. Those projects probably did not fail for being too ambitious. Fatigue, and changing interests are more realistic threats to works that take long too manifest themselves.

Fixing this was an inspiration, but inspiration also came from my own enthusiasm: The rediscovery of the trance-inducing graphics of the finest manic shoot-em ups.[1] The increasingly out-reaching exposure of the demoscene. And a feeling that it is now that we should think and act instead of leading our lives in the future tense, like todays mass culture, politics and work environment all seem to suggest.

It was time to join the choir again.

In September 2005 something emerged, product of many choices, some conscious and some other unconscious. *Clothing Like A T-shirt Saying I Wish*[2] was released then, on 2005.09.17, at the streamMega demo party, near the Finnish town of Tampere. With it came the following note:

> Clothing like a tshirt saying I wish [YOKO VERSION]
> !=
>
> 70050917 @1553 Ylöjärvi / Suomi
>
> _
>
> Barely two weeks of work. Intense and everything started from scratch. This intro is more oldschool than most so-called olschool demo are. Unoptimised and feeling yet no guilt about it. Wait for the TATE version.
>
> _

– Ce matin m'est arrivé un truc très désagréable
– Quoi donc?
– Je ne sais plus dire bonjour dans ma propre langue.
Pollution, oui. Chomage, opérations sur titre, harcelement sexuel, oui oui oui. Mais plus bonjour.
– Ah. Et moi j'en ai assez de cette foutue eau moderne qui ne fait plus de bruit quand elle bout.

–

Contact info: tpolm@tpolm.org

Why write about an intro? Why write this, you ask? Maybe because Clothing Like A T-Shirt Saying I wish, like its title suggest is very thin and fragile. It does not punch you in the face. Oh it may stay opaque, or exude a sort of confidence, but it just stands there, and I wanted to give it a body so it could be hurt by others. Another reason is that I wish more sharing of views might be done in written form in the demoscene. If it could inspire others to talk about *making* demos.

## 2 Description

Clothing Like A T-Shirt Saying I Wish is a sixty four kilobytes intro. Its duration is approximately ninety seconds, quite a bit shorter than usual intros and demos. It is strictly black and white, and the graphics and music are fully generated in real-time, two images excepted. (a TV on its side, and the label "!=")

This duration of ninety seconds is divided in parts of almost equal length, approximatively ten seconds long. Each part draws two-dimensional patterns in rhythm over an uniform vertical canvas.

The canvas has an aspect ratio of $width/height = 3/4$, in opposition to the wide-screen and large field of view formats much in use today. (16/9, 16/10) The two dimensional patterns make use of relatively simple mechanism: most of them appear to be made of moving objects (simple squares, arrows) moving along predefined paths, or interacting with each-other.

The only apparent signs of a macro-scale narrative[1] are firstly the rhythm established by the music which, although it does not seem to control the succession of parts, seems to rule the patterns themselves, the speed of elements, the appearances and causes and effects particular to the pattern itself. Secondly, on a wider scale, after forty-five seconds, the simultaneous appearance of a bass-drum, a kick, and the inversion of the colours between black and white. The canvas is afterwards uniformly black, and the moving objects white. The bass-drum also appear to be strongly linked to the patterns' movement, especially at the inversion point, where a more singular pattern is displayed: a not so uniform black canvas see two white-on-black images successively appear and dissolve again and again.

After this inversion, a succession of variations of the original patterns appear, until the end where the music and visuals both fade into the black canvas.

# Part II
# Design and Methodology

## 3 Where Am I

Imagine you are back on the fourth of September 2005. On that day, you decide to release something for the next demo-party in Finland, known as stream Mega, held from September 16th to September 18th. Imagine also that your previous effort took you three month to build, but that now you are contemplating at a mere thirteen days period. Nevertheless, you have a clear mental image of what you want to achieve. You don't know exactly how, but you know what. You remember the beautiful bullet patterns of the finest shoot-em ups. You want to contrast the often used cinematic 16/9 displays with an unusual

[1] for an abstract piece, by narrative we mean the manifestation of cause and effect

vertical display[3]. You remember how pleasant it is to work on how things move and interact, and less about how they precisely look.

With this in mind, the first decisive factor is the available time. With nine hours work days and a speculated eight hours of sleep a day to take into account, this leaves $13 days = 13*24-10*9-13*8 = 118 hours$ of "leisure" time, and if eating is required, I would say about $118-(13*2+2) = 90 hours$.

The initial image, that of a minimalistic piece made using two dimensional content comes both from the initial shoot-em up inspiration, and a pragmatic choice considering the remaining time. I was already working on a vertical shooting game machinery, enabling the creation of interactive and scripted particles on a two dimensional canvas, ruled by Newtonian physics, and with collision detection. It was decided to re-use this code-base taken into account it was certainly not mature enough.

The second factor thus becomes the use of a rigid system only capable of animating and displaying scripted or simulated particles on a canvas.

The third factor then turns out to be size, since although it was designed reasonably simple and small, this system was already a few kilobytes in size. It is reasonable to target the sixty-four kilobyte competition, granted that minimalism does not resonate well with a multi-megabyte form.

Additionally, in a way this was also inspired by Dierk "Chaos" Ohlerich's presentation at FMX/05, where he pushes the choice of programming content generation versus traditional production means not because it's cool, but because it's a sensible way to do so: generating both visuals and music seems one of the quickest way to achieve a coherent result.

For it is this coherency that gives confidence to a piece, contributing to its believability, and this is the final factor to take into account.

# 4 Where to go

Taking all this into account means that choices have to be made both to on what to achieve, and how to achieve it.

First, we have to balance what the piece shows with what it costs to do so. It should not be expensive to make, and taking the other factors into account, this leaves room only for a modest output. For the sake of coherency, it is also important to avoid any part to look more achieved than others. At the same time, there is a need to produce a reasonable impact.

Two-dimensional graphics are special because, for most of the time, we as human beings engage in the world in three dimensions. Our eyes are not at all like cameras or camcorders. Rather than experiencing our environment through pictures, we interact through a three dimensional perception. We do not interact with pictures directly, but through a complex reconstruction.

All of this to say that we cannot really engage with two dimensions only. Any picture goes through the machinery that determines our entire visual experience: our senses seem to complete the blanks, the empty spaces. They find alignments, weight shapes with others, group them by category. Even count them. With less things in the picture, the rest is filled by the spectator's experience.[2]

So is the obvious choice to keep a very streamlined two dimensional look, very abstract, and full of empty spaces? Only if we can play tricks with the spectator's perception, to build more out of less.

But we have to keep it coherent and balanced. If the visuals look outlandish, the audio also needs to equally sound abstract and drifty.

Scaffolding or building towers is out of the question. We need to start wide and low and grow a little step by step. But with a small quantity of material, and too wide a base, the mountain might also not look like any at all. It might not even look like a hill. So with few materials we must settle down on building a nice but very small hill, with an equally small surface.

We must be able to grow all things in parallel, with the same level of sophistication, the product always kept within the final limitations of size and well. It also had to keep working too.

[2] This is known as "Gestalt" or "Gestalt Theory" in psychology, but see also [4]

## 5 How to do it

To address the time limitation, I found necessary to adopt a regular development rhythm: working in sessions of a predetermined length[3] segmented into smaller periods[4], with a short pause in between and a longer pause between sessions. This division enables sessions to produce actual result, and periods make sure we can allow to regularly reflect on the work being done. Pauses are real pauses, preferably diverting your energy and intellect to something else or more mundane. [5] Each period corresponds to the realization of a more or less precise goal. A log is kept precisely, helping to know what's been worked on, and anything that needs to be worked on later. It's also important to always question the relative importance of individual tasks to the greater picture.

The whole work must still compile and work after each period. Reaching something deliverable after every few periods, and saving it for future comparisons is also worth it, if only to make sure we are staying within the size limits.

Half of the total time must be kept to work on the scripting and finishing touches. It is easy to forget all the details that make up a finished product.

Another thing before I leave the topic: sleeping is *not* optional.

Technically-wise, Writing the most expressive code[6] is essential. Expressivity: a good ratio between the number of lines of code dedicated to the actual result versus the number of lines dedicated to infrastructure. It enables dramatic changes to brought very quickly. The idea here is to make it possible to experiment while still controlling the process: We want to be able to completely alter a part while still being able to go back to the previous step. Keeping the feedback loop short between coding and analysing is important.

Aesthetically, some design choices were already provided by the initial image: a two dimensional canvas, with a vertical 3/4 aspect ratio. Swarms of particles moving in geometrical patterns, filling the screen with mazes in motion that an imaginary avatar might navigate in. Some basic concepts were chosen as base for patterns: Grouping and subdivisions, radiating patterns, sequences of tightly aligned objects. The idea of playing with the respective size of objects to introduce a fake perspective was also decided at this stage. Some other choices come from a certain coherence with more generic design choices[7]: no credits, and two parts with an abrupt transition in between.

The shape, the design of the actual product is thus a result of:

1. The limitations of the environment
2. The original concept or image
3. Chance

When considered as just a parameter of creation, limitations form a simplifying mechanism: they enable us to experiment on a limited, still manageable sub-set of parameters. They enable us not to be confronted with too vast possibilities, when we, either by chance or will, explore the bifurcations of the paths they offer.

The concept is the direction one pushes towards. It may not be easily translatable in an implementation, but should form a strong and lasting image.

[3] 180 minutes here

[4] 90 minutes here

[5] The most common objects make very good design exercises. We already have expectations about how they should work and what we can do with them. But at the same time through their current form they are the recipient of successful experiences, the limitations of their original form or technology, and the hidden hopes that men put into them.

Clocks are one of those common objects. Even if we rarely wear a watch anymore, almost all devices nowadays include a clock, though always the same digital clock. Gradually hidden and surrounding us, they lost their flamboyant shapes and end up looking all similar to each other.

If I was to design a clock, I would not make it at all like a digital watch. Digital watches are only meant to make sure we arrive on time to our meetings. They're not really saying something about time, more about communication and social structure. No, if I was to design a clock, it maybe would be a drum machine. Why did we build this hierarchy of years, months, weeks, days, hours, minutes and seconds? I believe it is because we at the same time represent the scales, the rhythms that our works requires. I believe our representation of time really is a representation of tasks, and the rhythm of these tasks inside other tasks.

[6] it may also help creating the most interesting bugs, more expressively.

[7] the != label

By chance, we mean what is happening day to day. Nothing ever goes in a straight direction, and this is where the creative mind plays its role. Creativity is what emerges as the remains of a collision with reality, the results of a form of problem solving.

# Part III
# Implementation

## 6 Rationale

The general idea I am playing with these days is very simple: first build a formal system or structure. Use it. Use it, then, try to subvert it until something different emerge. Why? What is being expressed here? Maybe I am trying to understand how creation works.

Maybe it does not really matter. The reactions of a spectator do not depend so much on the destination of our own expressions (what we meant) and even when they do, is it really the point? Some might want to include the spectator in the system, but it does not appear necessary. Spectators are already able to create new meanings and sensations out of a non-interactive form. *A contrario*, when given control of the expressions, I expect spectators to temporarily lose their critical abilities: because we are often unable to be critical of an experience of which we are the main actor. This surely explains why sometimes talking about a game is much more entertaining actually playing it.

You cannot build anything without materials, and a structure without means of expression becomes likewise meaningless. Graphics and sounds are the expression media of a demo, while code form its structure but at the same time its primary expression mean.

Of the $90 hours$ of leisure time our little calculation gave for thirteen days of work, it turns out I only could dedicate half of it for the implementation: $45 hours$. Until the end, where I also had to engage in furious air plane and party coding, on a laptop not even powerful enough to run the piece in realtime.

As always, I decided to make my work decently reusable, because this had to serve as a base for further works, one[8] of which was released only one month after, at the Spanish party BCN.

When prototyping, the different parts were made available through a menu instead of scripted as in the final work. The menu was made using the shoot-em up engine, the player's ship (which is otherwise invisible during the demo) acting as a mouse pointer emitting button activation events instead of devastating shots.

## 7 Materials

Graphically wise it was already pre-decided, in a way: two dimensional objects, rendering themselves over a canvas using OpenGL. As well as animating their rendering, they could also be animated as if they were solid, with a mass and shape. The effort was put into animating them together, their individual rendering considered secondary, some vanishing or appearances of objects excepted, and one single whole-screen effect was designed in the end. Some collision detection was even used for certain patterns, helped by partitioning the canvas in areas[5]. Movements were generally defined by the action of newtonian forces on solid objects (Verlet integration[6] was used).

This whole-screen effect appears as a sort of screen displaying images that dissolves almost immediately. Under the hood it models a non-homogeneous medium with varying temperature and phase. The phase determines whether a given area of the medium is liquid or solid, and differential equations rule how temperature changes when phases vary and how phase changes when temperature varies. Each area is then displayed as a black and white cell, depending on its phase, slightly modulated by the temperature. [7]

Audio wise, we had to have at least the basics. One could build a totally alternative synth mimicking the graphics' evolutions[8]. But this does not seem necessary nor even constructive, as music is one of the most striking example of an abstract yet mainstream form of expression. Representing structure through

[8] != - Walking on four

musically defined forms should prove more productive and coherent than as a figurative audio simulation.

The intention being to generate content from this structure I was talking about earlier, I had a very simple layer of scripting put inside the system. What do I mean by scripting? I am talking about a method to create "events", points in time with singular properties. Causes to consequences. They may be followed by abrupt changes, or gradual progressions, and one of the goal is to make sure audio and visuals can listen and react when any of these appear.

# 8 How to build a synth in two days.

I had set my mind on fully generating the music. For the actual audio production, subtractive synthesis is always a safe choice. It produces sounds that most are accustomed with, and does not require a big up front effort in terms of design and implementation. The minimum set of material was thus determined to be: oscillators, envelope generators, filters.

For oscillators, I chose to implement band-limited oscillators for the square and sawtooth oscillators. A band-limited implementation guarantees that no harmonics are produced above what the sample rate allows, which means no aliasing. The square and sawtooth waveforms were created from Emanuel Landeholm's implementation of a band limited sawtooth wave form. [9] The square wave form was created by integrating a combination of positive and negative pulse trains as proposed in [10] No pulse-width control was added.

Another necessary element is a low pass filter. A standard choice is a so-called Butterworth (biquad) filter, which provides a 12db slope and a resonance. Not as sexy as a moog filter, but it will do the job.

To control the parameters a classic ADSR[9] envelope was made. Delay lines also, as a basic block. The programming interface to them can be made very simple: a buffer (delay line) in which only one producer can write (the writer) and from which several consumers read at different intervals from the current time.

The rhythm was picked as 112.5 beats per minute.

## 8.1 Drums

From those elements, we can first build a simple bassdrum.(figure 2) I chose to accompany this bassdrum, with the simplest possible percussive instrument: a short, tonal metronome-like (figure 3) sound "tick".

A syncopated, alternating rhythm was picked:

click [every 2 ticks]
bassdrum [every tick + offset[3/4 of a tick] ]

## 8.2 Strings

This alone feels quite empty, so we needed something to fill the spectrum a bit. Synthetic string like sounds would do the trick, but with only an oscillator, even if layered as chords, it would sound quite dry. At this point I remembered having stumbled on an orchestra[10] by Sean Costello for the CSound synthesizer programming language back in 1996-1997. The idea of his "Stringphaser" orchestra was to emulate the liquid, powerful chorus effect of analog machines. ("String ensembles") It is nowaday pretty hard to find the original orchestra on the net, but I could find an archive of it in [11].

The goal was to drive a polyphonic, very simple synth made with just an oscillator and an envelope generator through this "string phaser" in order to turn it into something decent for atmospheric sounds.

The algorithm of the stringphaser itself is to initialise three delay lines, each modulated by two – slow and fast – groups of three different waveforms, rotated $2\pi/3$ one to another. This part sounds a bit like a vibrato, and helps create the chorusing effect. The result of these delay lines is then taken to a stack of allpass filters fed back to themselves, and then mixed together. Through this we obtain a mix of flanging/chorusing and vibrato, making the sound more lively and full.

[9] Attack Delay Sustain Release

[10] this is how synthesizer constructs are named in CSound

The output of the stringphaser then goes through a reverb, which is implemented as a simple version of a Feedback Delay Network (FDN) reverb.[12]

The big picture (figure 4) can be broken down into:

- a phaser part (figure 5) with $n = 5$ and $coeff = -0.3$
- a vibrato part (delay lines) figure 6
- a reverb (figure 7) where delay1 is $73.06 + 1.2 * noise[at 3100 hz] milliseconds$, delay2 is $83.9 + 1.32 * noise[at 3500 hz] milliseconds$, delay3 is $97.7 + 1.87 * noise[at 1110 hz]$, delay4 is $107.3 + 0.66 * noise[at 3973 hz]$. The multitap delay is setup per figure 8.

Although it is quite power hungry, since samples are processed one after another instead of blocks of samples per blocks of samples, this computational power does not depend on the polyphony of the synth that is being driven through the effect.

We then drive this polyphonic string ensemble by a stream of notes taken from a very simple algorithm:

$$\text{a note [every 8 ticks, 6 seconds long] with } \\ midinote = 45 + random[0 \cdots 16]$$

# 9 Causes and consequences

The piece does not show very complex interactions, but as it was created as a stepping stone for more interesting works, I took some time to think about how to best integrate audio and video events from a bottom-up perspective. My main motivation being that one side should not exercise too much control on the other.[11] Relations shall be created between the particular rhythm or evolution of one graphical effect, and the evolution of the music not by synchronising one to another, but by controlling both via a separate, dedicated layer.

[11] Which reminds me what a well known amiga scener once said to me, by the bonfire of breakpoint'2004. I'll try to paraphrase it here: in most pc demos, it's the music that decides what and when things happen, while in amiga demos, it's the graphics that do.

Akin to such graphical languages like MAX[13], Pure Data[14] or VVVV[15] one immediately envisions a network-like structure. But I do not desire a component-based description. I hope that higher abstractions could be built to describe particular patterns of interaction, something which is difficult to do in such graphical languages.

Usually video or audio effects are implemented on a template similar to this:

```
class Effect {
  public:
    /*
     * called frame by frame, to
     * create the audio or
     * visuals
     */
    processFrame (double ms, ...);

    /*
     * a parameter, that will be
     * used when processing the
     * next frame.
     */
    void setParameter1 (float p);
}
```

First it is impossible to make the parameter vary during a frame, and secondly, parameter variations require a timeframe or sequencer. Also, common things like parameter interpolations or on the contrary, quantisations must be implemented inside the effect.

I replaced this common pattern with the following:

```
/*
   An object representing a
   connection to sources of
   events (of type T)

   By extension its
   implementation determines
   a way the environment
   changes with time.
 */
template <typename T>
  class Actuator;
```

```
/*
   we can subscribe to an actuator,
   and we thus obtain a type-safe
   source of values.
 */
template <typename T>
  Source<T> subscribe (Actuator<T>& a);

/*
   a source is a way to
   tap-in, to obtain
   values out of a
   temporal motif
 */
template <typename T>
class Source {
  /*
   * will return false occasionally
   * in case of discontinuous
   * sources.
   */
  bool hasValue (double ms);
  /*
   * returns the realtime value
   */
  T getValue (double ms);
}

class Effect {
  public:
    /*
     * called frame by frame, to
     * create the audio or
     * visuals
     */
    processFrame (double ms, ...);

    /*
     * sets the parameter's
     * actuator, which will
     * be used when
     * processing the next
     * frame.
     * The effect will
     * usually subscribe
     * to it immediately.
     */
    void setParameter1
      (Actuator<float>& pa);
}
```

Then, in Effect :: processFrame, the effect obtains realtime values from the source returned when subscribing to the actuator.

The parameter variations are then built totally externally from the effect itself. As an example, we have built the following actuators:

- an OscillatorActuator turning any audio oscillator into a source of events, useful when creating LFOs.
- an ADSR actuator implementing the usual Attack Decay Sustain Release envelope.
- ConstantActuator returning a constant value either once or repeatedly.
- MetronomeActuator returning a value at given intervals.

We can then build more complex actuators by combining simpler ones via combinators:

- a combinator that enables us to addition, multiply the result of actuators together.
- a logical operation actuator combinator that returns the value from the first actuator that returns a value.
- a combinator that returns the value of an actuator only if the other fires. This one is used for example to sample an actuator at a given frequency.

A difficulty arises when one must implement the actuator themselves, as they may be used by effects that run at different frequencies, and in different threads. We would not want side-effects to arise from the fact an actuator is beeing used by two objects at the same time.

This is the reason for the Actuator/Source dichotomy I introduced: while the Actuator may itself

be shared between all the audio/visual objects, the Source is held by only one object. It can thus store a private state, dependent for example on which values have been returned so far.

Since actuators exist outside effects, their lifespan must be controlled globally. To this end we chose to address them via smart pointers, reference counted pointers allowing to properly deallocate them when they are not referenced to anymore.

## 10 Code reuse via Mixin-style programming

When confronted with the design of the two dimensional patterns I sought to create, I set a goal to implement the different particle behaviours in the minimum amount of code, while at the same time when prototyping being able to add and remove behaviours at will. It felt important to be allowed to try out different versions of the same effect, or change the basic behaviour of particles, the way they accelerates or moves, their lifetime or even make them spawn new particles.

Usually a behaviour is a combination of simpler, disjoint behaviours: we should be able to express interesting patterns in terms of combinations of specific behaviours, implementable in a generic way: the base class should not matter when writing the behaviour's code, except that it must adhere to a certain vague contract. This contract was that all the objects had to be able to take part in a physics simulation.

Composing behaviours like that is very easy in a dynamic language such as Ruby where you can import multiple classes as a way to merge and inherit all of their specificities. It is very easy there, because unlike c++, the method lookups are done at runtime, and their exact "location" inside another class or type is not taken into account at all. This style of programming has been named "Mixin-based."

Although it is not a very common idiom in c++, we can reach this goal by composing c++ templates together, as a way to create new classes as a combination of roles. [16] see also figure 1

The Mixin style enables good flexibility, through a form of late "binding". Choices are deferred until the compiler instantiates the template and its functions. So you can reference symbols that are still not implemented, as long as the methods or classes are not used the code still compile as a whole. One can quickly turn an object into something else by composing ready-made behaviours:

AirShip becomes TimeLimited< AirShip > and voila, it can automatically disappear after a while without having to change the original AirShip class.

One can thus combine archetypes together to make more complex behaviours. For example: ComposedObject< KeptInsideScreen< AirShip > > defines an object that is composed of sub-objects, with a constraint that it cannot leave the screen's area, and as an airship, is subjected to friction from air and can only move through the acceleration of its engines.

Some issues arise with this style of programming: construction of objects becomes a bit unnatural, as one must find a common protocol to build new instances for what may be completely different subtrees of the class hierarchy. Some work[17] exist to solve the issue, but we choose to implement it via a constructor parameter class, a description for new instances that gets inherited from one level to another, instead of via template meta-programming. Another serious issue when coding a sixty-four kilobyte intro is the resulting "code bloat" As new combinations of behaviours get created, the compiler creates entirely new classes, new functions. Of course using a good packer[18] helps, but this is an issue to watch out: if many different combinations are used, then it becomes interesting to turn the Mixin-style code into a more compact form, after prototyping.

# Part IV
# Conclusion

While the result of these thirteen days of work was received in an interested although not-quite-enthusiastic way, I am rather happy with it. It is particularly special to me because it is modest and

```
/*
 * This behaviour tracks instantiations of the class it is added to.
 */
template <class Base>
class AllocationTraced : public Base {
public:
        /*
         * notice the constructor: here we have to define a standard
         * construction protocol, to be able to chain all the mixins
         * templates until we reach the different base class' constructors.
         */
        AllocationTraced (const Base::InstanceDesc& description) :
          Base (description) {
              allocationId = newAllocation ();
              increaseAllocationCounter();
        }

        virtual ~AllocationTraced () {
              decreaseAllocationCounter();
        }
private:
        unsigned int allocationId;
};

// (...)

class String {
  public:
  struct InstanceDesc {
    InstanceDesc (const char s) : s(s) {}
    const char* s;
  };

  String (const InstanceDesc& desc) {
  // ...
  }
}

class Int {
  public:
  struct InstanceDesc {
    InstanceDesc (int i) : i(i) {}
    int i;
  };

  Int (const InstanceDesc& desc) {
  // ...
  }
}

// (...)

int main (int argc, char** argv) {
  const Int::InstanceDesc iDesc (0xDEADBEEF);
  auto_ptr<Int> i = new AllocationTraced<Int> (iDesc); // allocation counter = 1

  const String::InstanceDesc sDesc ("a random string")
  auto_ptr<String> s = new AllocationTraced<String> (sDesc); // allocation counter = 2

  // destructors get called then allocation counter = 0
}
```

Figure 1: a mixin example, tracing allocations of the classes it is mixed-into

not so impressive. Which is fine, even if or maybe because it's not very demo-like.

Bringing to life simple two-dimensional patterns, was also quite fun, in a nostalgic kind of way. And sharing this with others too.

# Part V
# Appendices

## 11 Drums

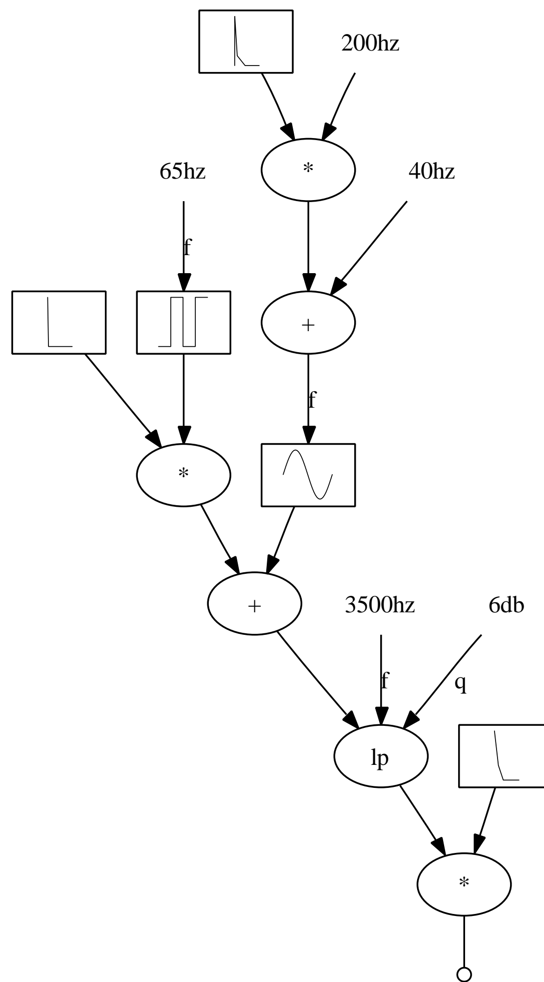


Figure 2: A basic bassdrum implementation

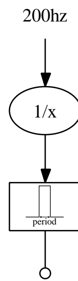


Figure 3: A short high-pitched click

# 12 String phaser

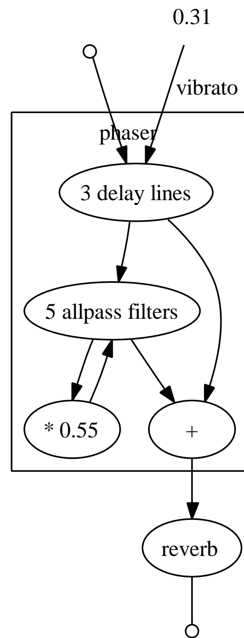


Figure 4: Global view of the stringphaser

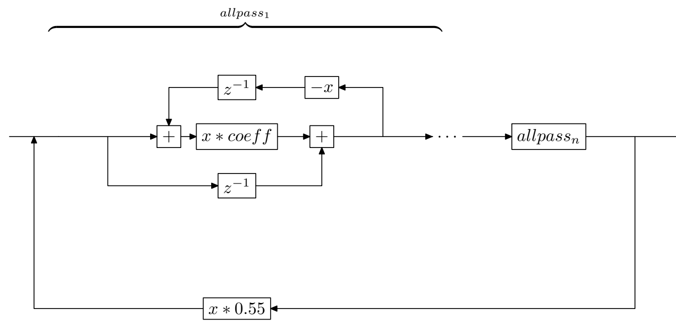


Figure 5: Phaser

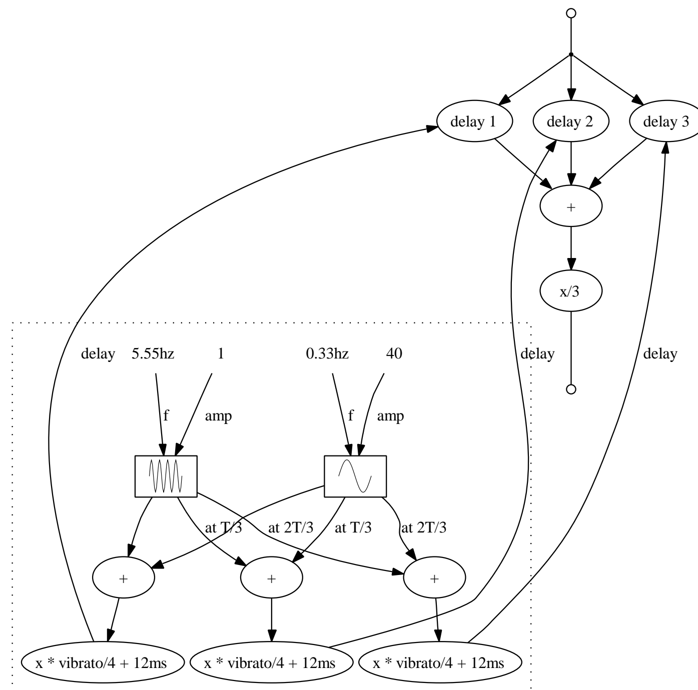


Figure 6: Vibrato made using varying delay lines

# 13 FDN Reverb

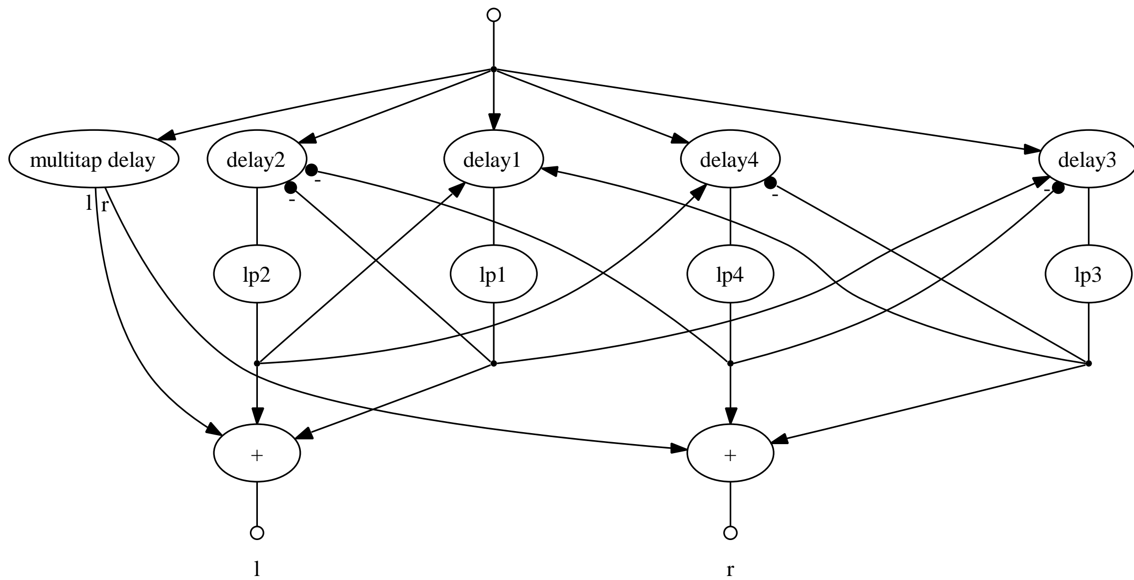


Figure 7: Feedback delay network reverb

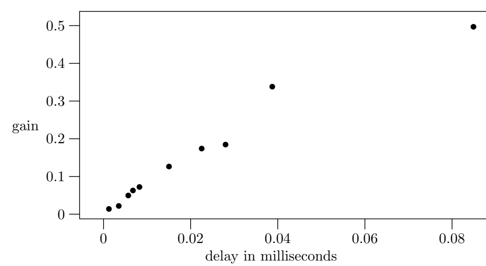


Figure 8: Multitap delay used in the reverb

# References


[1] Guwange, E.S.P. Galuda, Dodonpachi, Dangun Feveron are among the many acclaimed vertical shoot-em ups (STG) that the japanese studio Cave has released throughout the years. `http://www.cave.co.jp/`
They innovated in the “Manic” shooter genre, characterized by screen-filling bullet patters, and an emphasis on dodging by navigating through these moving mazes. (Another emphasis exists, on scoring mechanisms)

[2] != - Clothing Like A T-shirt Saying I Wish
`http://scene.org/dir.php?dir=/parties/2005/stream05/in64/`
`http://www.pouet.net/prod.php?which=19028`

[3] The Fifty-Three Stations of the Tokaido by Ando Hiroshige, Ukyo-e elegance of the vertical form

[4] “Point and Line to Plane” by Wassily Kandinsky

[5] `http://en.wikipedia.org/wiki/Quadtree`

[6] Verlet integration
`http://en.wikipedia.org/wiki/Verlet_integration`

[7] ”Visual Simulation of Ice Crystal Growth” by Theodore Kim and Ming C. Lim
`http://www.cs.unc.edu/~geom/ICE/small.pdf`

[8] _{_ by the czech group Downtown, released in 2001.
`http://downtown.dee.cz/`

[9] Bandlimited synthesis of sawtooth (Emanuel Landeholm, March 2002)
`http://www.musicdsp.org/showArchiveComment.php?ArchiveID=90`

[10] Alias-Free Digital Synthesis of Classic Analog Waveforms by Tim Stilson and Julius Smith

[11] The CSound Book, Edited by R.Boulanger, 2000 The MIT Press. ISBN 0-262-55261-6

[12] Sean Costello cites “Designing Multi-Channel Reverberators,” by John Stautner and Miller Puckette in Computer Music Journal, Vol. 6, No. 1, Spring 1982, P.52-65.

[13] A product sold by cycling74, first conceived at IRCAM by Miller Puckette for musical applications. `http://cycling74.com/products/maxmsp`

[14] Pure Data, an opensource MAX/MSP like language with a rather active community. Created by Miller Puckette
`http://iem.at/pd/`
`http://www-crca.ucsd.edu/~msp/`

[15] VVVV by the german company meso, a video synthesis package inspired by MAX.
`http://vvvv.meso.net/`

[16] “Mixin-Based Programming in C++” by Yannis Smaragdakis and Don Batory

[17] "A Solution to the Constructor-Problem of Mixin-Based Programming in C++" by Ulrich W. Eisenecker, Frank Blinn, and Krzysztof Czarnecki

[18] 20to4 a packer by Marcus Winter (Muhmac / Freestyle)
`http://20to4.net/`