THE DESIGN OF A LIGHTWEIGHT DSP PROGRAMMING LIBRARY

Victor Lazzarini

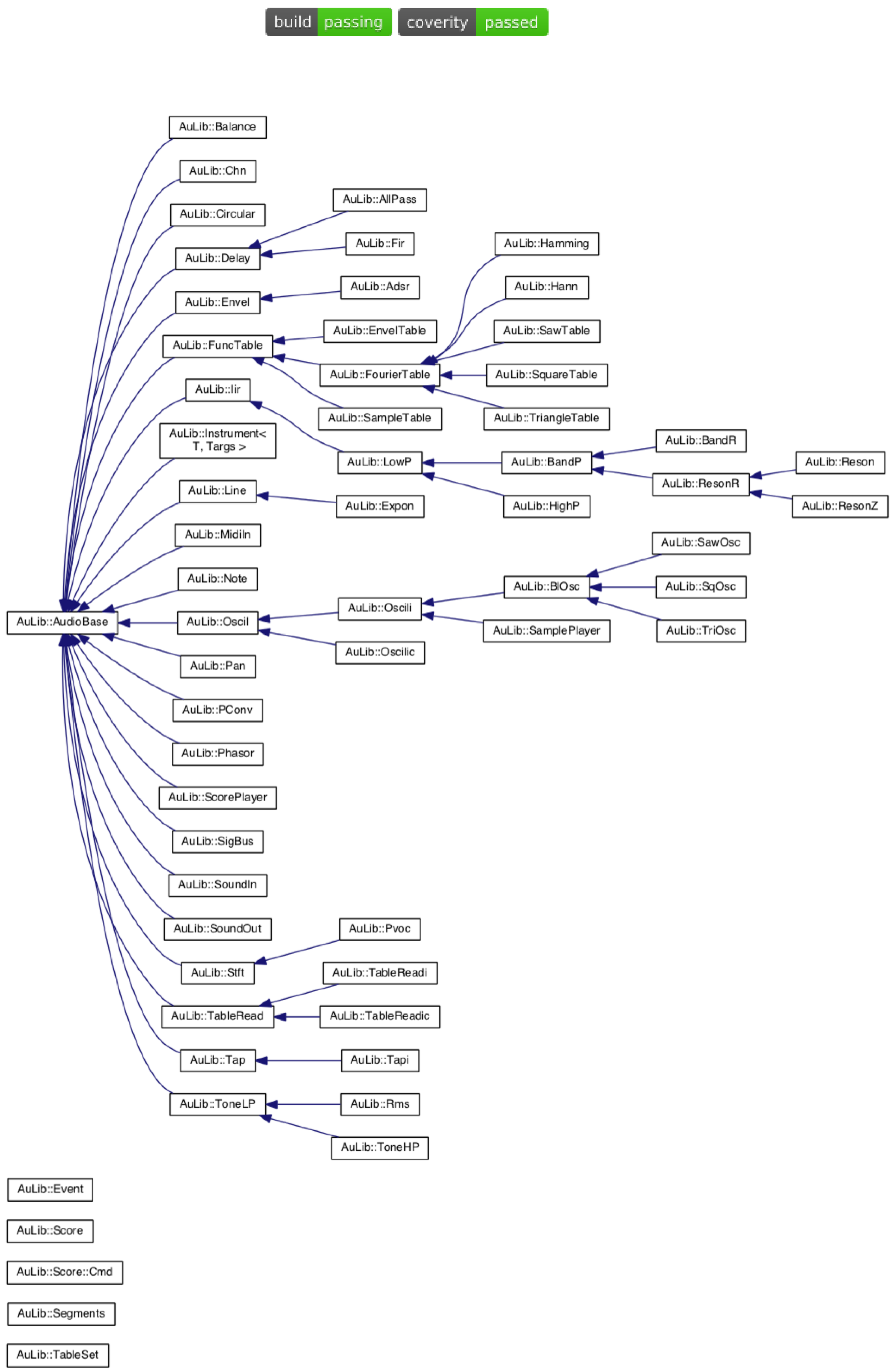
Music Department
Maynooth University
Maynooth, Co Kildare
Ireland

victor.lazzarini@nuim.ie

Abstract

This paper discusses the processes involved in designing and implementing an object-oriented library for audio signal processing in C++ (ISO/IEC C++14). The introduction presents the background and motivation for the project, which is related to providing a platform for the study and research of algorithms, with an added benefit of having an efficient and easy-to-deploy library of classes for application development. The design goals and directions are explored next, focusing on the principles of stateful representations of algorithms, abstraction/ encapsulation, code re-use and connectivity. The paper provides a general walk-through the current classes and a detailed discussion of two algorithm implementations. Completing the discussion, an example program is presented.

https://aulib.gitub.io



AuLib class hierarchy

```cpp
// Sinewave voice, derived from AuLib Note class
class SineSyn : public Note {
protected:
  // control list
  uint32_t m_atn, m_dcn, m_ssn, m_rln;
  std::map<uint32_t, double> m_ctl;
  double m_bend;

  // signal processing objects
  Adsr m_env;
  Oscili m_osc;

  // DSP override
  virtual const SineSyn &dsp() {
    if (!m_env.is_finished())
      set(m_osc(m_env(), m_cps * m_bend));
    else
      clear();
    return *this;
  }

  // note off processing
  virtual void off_note() { m_env.release(); }

  // note on processing
  virtual void on_note() {
    m_env.reset(m_amp * 0.2, m_ctl[m_atn] + 0.001, m_ctl[m_dcn] + 0.001,
                m_ctl[m_ssn] * m_amp, m_ctl[m_rln] + 0.001);
  }

  // msg processing
  virtual void on_msg(uint32_t msg, const std::vector<double> &data,
                      uint64_t tstamp) {

    // pitchbend;
    if (msg == midi::pitchbend) {
      int32_t bnd = (int32_t)data[1];
      bnd = (bnd << 7) | (int32_t)data[0];
      double amnt = (bnd - 8192.) / 16384.;
      m_bend = std::pow(2., (4. * amnt) / 12.);
    }
    // ctrls: att, dec, sus, rel
    else if (msg == midi::ctrl_msg) {
      uint32_t num = (uint32_t)data[0];
      m_ctl[num] = data[1] / 128.;
    }
  };

public:
  typedef std::array<int, 4> ctl_list;

  SineSyn(int32_t chn, SineSyn::ctl_list lst)
      : Note(chn), m_atn(lst[0]), m_dcn(lst[1]), m_ssn(lst[2]), m_rln(lst[3]),
        m_ctl({{m_atn, 0.01}, {m_dcn, 0.01}, {m_ssn, 0.25}, {m_rln, 0.01}}),
        m_bend(1.),
        m_env(0., m_ctl[m_atn], m_ctl[m_dcn], m_ctl[m_ssn], m_ctl[m_rln]),
        m_osc() {
    m_env.release();
  };
};

// Sawtooth voice, derived from SineSyn
class SawSyn : public SineSyn {
  SawOsc m_saw;

  // DSP override
  virtual const SawSyn &dsp() {
    if (!m_env.is_finished())
      set(m_saw(m_env(), m_cps * m_bend));
    else
      clear();
    return *this;
  }

public:
  SawSyn(int chn, SineSyn::ctl_list lst) : SineSyn(chn, lst){};
};
```

```cpp
// Convolution reverb
class Reverb {
  const uint32_t esmps = 32;
  const uint32_t partsmps = 1024;
  SampleTable m_ir;
  Fir m_early;
  PConv m_mid;
  PConv m_tail;

public:
  Reverb(const char *impulse)
      : m_ir(impulse, 1), m_early(m_ir, 0, esmps),
        m_mid(m_ir, esmps, 0, esmps, partsmps), m_tail(m_ir, partsmps, 0, partsmps){};

  const AudioBase &operator()(const AudioBase &in, double g) {
    m_tail(in);
    m_mid(in);
    m_tail += m_mid += m_early(in);
    m_tail *= g;
    return m_tail += in;
  }
};

static std::atomic_bool running(false);
void stop_synth(){
  running = false;
}

// MIDI synthesiser
// ir - reverb impulse response file name
// dev - MIDI device number
// attn, decn, susn, reln, revn - midi ctl numbers
int midi_synth(const char *ir, int dev,
               int attn, int decn, int susn, int reln,
               int revn) {
    // Sinewave Synthesizer - channel 0 (MIDI 1), 8 voices
    Instrument<SineSyn, SineSyn::ctl_list> sinsynth(8, 0, {{attn, decn, susn, reln}});
    // Sawtooh Synthesizer - channel 1 (MIDI 2), 8 voices
    Instrument<SawSyn, SineSyn::ctl_list> sawsynth(8, 1, {{attn, decn, susn, reln}});
    // Convolution reverb
    Reverb reverb(ir);
    // Audio & MIDI IO
    SoundOut out("dac", 1, 128);
    MidiIn midi;

    if (midi.open(dev) == AULIB_NOERROR) {
       running = true;
       // listen to midi on behalf of sinsynth & sawsynth
       while (running)
          out(reverb(midi.listen(sinsynth, sawsynth), midi.ctlval(-1, revn)));
    } else running = false;
    return 0;
}
```

MIDI synthesizer example, using the **`Instrument`** class for polyphonic performance. Voices are modelled by **`Note`**-derived classes **`SineSyn`** and **`SawSyn`**, which are instantiated by the **`Instrument`** objects `sinsynth` and `sawsynth`, each one responding to a specific channel and controller list.

These instruments are passed to a `listen()` method of the **`MidiIn`** object `midi`, so that they can be controlled via MIDI. Each instrument holds the mix of its active voices. The output of `midi`, which contains the mixed audio of all instruments, is sent to the **`Reverb`** object `reverb` and from there to the output.

Although this example shows a MIDI-based application, the same principles can be applied to other means of control (e.g. Osc, score languages, etc.).

SMC2017, Sound and Music Computing 2017, 5-8 July 2017, Espoo, Finland