Units of Measurement

https://github.com/Sinecure-Audio/UnitsTalk

Mark Jordan-Kamholz

- I’m a musician who got hooked on making audio software in college.
- I design and release plugins, mainly through my company Sinecure Audio.
- I’m currently based in Berlin, where I’ve recently been writing sound routines in 6502 assembly for Robert Henke’s 8032av.

What is a unit?

What is a unit?

A standard measure of a quantity.

What is a unit?

A standard measure of a quantity.

???

Why would you use unit types?

There are more reasons than you might think!

Why would you use unit types?

- Clearer API semantics
- Type safety
- Automatic conversion of underlying value when assigning an object of one type to a different type.
- Consolidation of conversion code.

How do we implement units?

- There are several existing solutions:
  - Boost.units, UNITS, etc.

How do we implement units?

- There are several existing solutions:
  - Boost.units, UNITS, etc.
  - Juce has a Decibel class, what about extending that?

How do we implement units?

SinecureAudio/Units

- Single Header
- Simple CRTP base class that holds a value, and defines all arithmetic and bit-shifting operators (which operate on the underlying type)
- Derived Classes inherit from this and define conversions
- Derived Classes can re-implement operator overloads for different functionality
- Several different specializations for Units that deal with loudness and filter resonance

Decibel Unit

```
1   template <typename NumericType>
2   class Decibel : public Unit<Decibel, NumericType>
3   {
4   public:
5       constexpr Decibel() = default;
6       template <typename T>
7       Decibel(const Decibel<T>& other) {this->value = NumericType(other.count());};
8       explicit Decibel(const NumericType& initValue) { this->value = initValue; };
9
10      Decibel(const Amplitude<NumericType>& amplitudeValue)
11      {this->value = convertAmplitudeToDecibel(amplitudeValue.count());}
12      Decibel& operator=(const Amplitude<NumericType>& amplitudeValue)
13      { this->value = convertAmplitudeToDecibel(amplitudeValue.count()); return *this; };
14
15      auto convertAmplitudeToDecibel(const NumericType& amplitudeValue,
16                                     const NumericType& minusInfinitydB = defaultMinusInfinitydB) {
17          return std::max(minusInfinitydB, std::log10(amplitudeValue)* NumericType{20});
18      }
19      //...
```

Decibel Unit

```
1   const Amplitude<float> amplitude = .5f;
2   const Decibel<float> decibel = amplitude;
3
4   const Decibel<double> halfPowerDecibel = Amplitude<double>{std::sqrt(.5)};
5   const Amplitude<double> halfPowermplitude = halfPowerDecibel;
6
7   std::cout << decibel << '\n';// outputs -6.0206dB
8   std::cout << halfPowermplitude << '\n';// outputs .707107
9   std::cout << halfPowerDecibel+halfPowerDecibel << '\n';// outputs 0dB
10  std::cout << -20.0_dB << std::endl;// outputs -20dB
11  std::cout << -200.0_dB << std::endl;// outputs -inf dB
```

https://godbolt.org/z/AR-65-

Decibel Unit

```
1   template <typename T, typename U>
2   constexpr auto operator+(const Decibel<T>& lhs, const Decibel<U>& rhs){
3       using CommonType = std::common_type_t<T, U>;
4       return Decibel<CommonType>{10.0* std::log(std::pow(10.0, lhs.count()/10.0) +
5                                                 std::pow(10.0, rhs.count()/10.0))};
6   }
7
8   template <typename T, typename U>
9   constexpr auto operator*(const Decibel<T>& lhs, const Decibel<U>& rhs){
10      using CommonType = std::common_type_t<T, U>;
11      return Decibel<CommonType>{lhs.count() + rhs.count()};
12  }
```

Some Practical Examples

- Envelopes
- Filters
- Units as Parameters

Envelope Timing

Seconds or milliseconds?

Envelope Timing

Both?

Units of Time

Units of Time

```
std::chrono::seconds timeInSeconds{3.2};
// Doesn't compile!
// You can't assign a floating point chrono unit or value to an integral chrono unit...
```

Units of Time

```
1   // represents 3.2 seconds;
2   std::chrono::duration<float> timeInFractionalSeconds{3.2f};
3
4   // represents 3.2 milliseconds
5   std::chrono::duration<double, std::milli> timeInFractionalMilliSeconds{3.2};
6   // also represents 3.2 milliseconds
7   std::chrono::duration<double, std::ratio<1, 1000>> timeInRatioMilliSeconds{3.2};
8
9   // 3.2 deca seconds, which is 32 seconds
10  std::chrono::duration<double, std::deca> timeInDecaSeconds{3.2};
11  // 3.2 kilo seconds, which is 3,200 seconds;
12  std::chrono::duration<double, std::kilo> timeInKiloSeconds{3.2};
13
14  // can convert between different duration lengths
15  std::chrono::duration<double> manySeconds = timeInKiloSeconds;
16
17  // prints 3,200
18  std::cout << manySeconds.count() << std::endl;
```

https://godbolt.org/z/EECnpQ

Envelope Timing

```
1   template<typename NumericType>
2   class ADSR
3   {
4       //you need to store the samplerate to convert samples to seconds when envelope time changes.
5       void setSampleRate(double spec);
6
7       //set the length of an envelope segment.
8       void setAttackTime(const std::chrono_duration<NumericType, std::milli>& newAttackTime);
9       void setDecayTime(const std::chrono_duration<NumericType, std::milli>& newDecayTime);
10      void setReleaseTime(const std::chrono_duration<NumericType, std::milli>& newReleaseTime);
11
12      //set sustain gain
13      void setSustainGain(const Decibel<NumericType>& newSustainGain);
14
15      //generate the envelope
16      NumericType perform();
17      //...
```

Resonance and Q

Resonance and Q Units

```
1   template <typename NumericType>
2   struct ResonanceCoefficient : public Unit<ResonanceCoefficient, NumericType>
3   {
4       ResonanceCoefficient() = default;
5       explicit ResonanceCoefficient(const NumericType& initValue)
6       {this->value = initValue;}
7
8       template <typename T>
9       ResonanceCoefficient(const ResonanceCoefficient<T>& other)
10      { this->value = NumericType{other.count()}; };
11
12      ResonanceCoefficient& operator=(const QCoefficient<NumericType>&& newCoefficient)
13      { this->value = convertQToResonance(newCoefficient.count()); return *this; };
14
15      static constexpr NumericType convertQToResonance(const NumericType& Q) {
16          return NumericType{1.0} - NumericType{1.0} / (NumericType{2.0} * Q);
17      }
18  };
```

Resonance and Q

```
1   template<typename NumericType>
2   struct SVFFilterParams : public dsp::StateVariableFilter::Parameters<NumericType>
3   {
4       void setCutOffFrequency(const double& sampleRate,
5       const NumericType& frequency,
6       const QCoefficient<NumericType>& Q = QCoefficient<NumericType>(1.0 / sqrt(2.0))) {
7           dsp::StateVariableFilter::Parameters<NumericType>::
8               setCutOffFrequency(sampleRate, frequency, Q.count());
9       }
10
11      void setCutOffFrequency(double sampleRate,
12          NumericType frequency,
13          NumericType resonance = static_cast<NumericType>(1.0 / sqrt(2.0))) {
14          jassertfalse;// Use units for your resonance parameter instead of numeric types!
15      }
16  };
```

Resonance and Q

```
1   ReferenceCountedObjectPtr<SVFFilterParams<double>> svfFilterParams = new FilterParams<double>;
2   dsp::StateVariableFilter::Filter<double> svfFilter(filterParams.get());
```

Units and Parameters

Why would I use unit types when all of my parameters are bools, ints, floats, or strings?

Units and Parameters

```
1   using ResonanceParameter = AudioParameterUnit<ResonanceCoefficient<float>, float>;
2   using SecondsParameter = AudioParameterUnit<std::chrono::duration<float>, float>;
3   using DecibelParameter = AudioParameterUnit<Decibel<float>, float>;
4
5   state (*this, nullptr, "state",
6   {
7   std::make_unique<ResonanceParameter>("resonance",
8                                        "Filter Resonance",
9                                        NormalisableRange<float>(0.01f, 1.0f, .001f), 0.01f),
10
11  std::make_unique<SecondsParameter>("attack",
12                                     "Attack",
13                                     NormalisableRange<float>(0.0f, 10.0f, .001f, 1.0f/3.2f), .02f)
14
15  std::make_unique<DecibelParameter>("sustain",
16                                     "Sustain",
17                                     NormalisableRange<float>(-120.0f,  0.0f, .01f),  0.0f)
18  })
```

Units and Parameters

```
22  ADSR::Parameters adsrParameters;
23
24  adsrParameters.attack =  *state.getRawParameterValue("attack");
25  adsrParameters.decay =   *state.getRawParameterValue("decay");
26  adsrParameters.release = *state.getRawParameterValue("release");
27
28  const Decibel<float> sustainGain{*static_cast<DecibelParameter*>(state.getParameter("sustain"))};
29  adsrParameters.sustain = Amplitude<float>{sustainGain}.count();
30
31  float cutoffParameterValue{300.0f};
32  ResonanceCoefficient<float> resonanceParameterValue{0.01f};
33
34  cutoffParameterValue = *static_cast<AudioParameterFloat*>(state.getParameter("cutoff"));
35  resonanceParameterValue = *static_cast<ResonanceParameter*>(state.getParameter("resonance"));
```

The End!

The End!

Questions?