DOPPLER EFFECTS WITHOUT EQUATIONS

Peter Brinkmann, [*]
peter.brinkmann@gmail.com

Michael Gogins,
Irreducible Productions
michael.gogins@gmail.com

ABSTRACT

We present a fast and robust method for approximating sound propagation in situations where audio and video frame rates may differ significantly and positions of sound sources and listeners are only known at discrete times, so that numerically stable velocities are not available. Typical applications include 3D scenes in virtual environments where positions of sources and listeners are determined in real time by user interaction. Our method employs a computationally inexpensive heuristic that converges to the exact solution for constant speeds and achieves convincing Doppler shifts in general.

1. INTRODUCTION

We want to simulate the propagation of sound in a 3D scene in a virtual environment. More formally, we have a listener with position $q(t)$ at time t (measured in samples at sample rate f_0) and a sound source with position $p(s)$ at time s. Sound travels at a finite speed c, so that the time t at which the listener hears a sound and the time s at which the sound was emitted are related by the equation

$$s = t - \gamma d(p(s), q(t)), \tag{1}$$

where $d(p(s), q(t))$ is the distance and $\gamma = \frac{f_0}{c}$ is the number of samples per distance. This is the basic Doppler identity, and differentiation with respect to t yields the familiar frequency shift formula (see, for example, [1]):

$$f = \frac{c + v_l}{c + v_s} f_0, \tag{2}$$

where v_l is the velocity of the listener and v_s is the velocity of the source.

So, in order to compute the sample heard at time t, we merely have to solve Equation 1 for s and compute the sample emitted at time s. Up to interpolation and possibly some filtering, we are done.

Or are we?

We need to deal with several complications: Firstly, Equation 1 is nonlinear, so that solving it may be computationally expensive. Secondly, and more importantly, we don't actually know $p(s)$ or $q(t)$ at all times; we merely know those positions at discrete points in time, so that we need to interpolate. Even worse, those positions will generally be updated at the video frame rate, and the typical time between video frames is an eternity from the point of view of audio. As far as the audio is concerned, source and listener just sit around for a long time, then they jump discontinuously, then they sit around, etc. Hence, we need to find a suitable interpolation scheme for source and listener positions, and then we need to solve Equation 1 for this interpolation.

In spite of all that, we have found an approximate solution that works well in practice. It was born out of the intuition that if we have to fudge source and listener positions between video frames, then we might just as well fudge things altogether. Our method produces subjectively convincing Doppler effects without the need to solve equations. Moreover, it is robust, computationally cheap, and straightforward to implement.

2. RELATED WORK

Many libraries and software packages (e.g., OpenAL) implement some version of Doppler shifts. However, all solutions that we are aware of are based on Equation 2 rather than Equation 1, i.e., they require the velocity of source and listener to be known. In the settings that we are interested in, such as interactive virtual environments, source and listener positions are only known at relatively sparse points in time, so that numerically stable velocities are not available.

Moore's space unit generator [1] approximates Doppler effects with an interpolated variable-length delay line, as does our method. The crucial difference, however, is that we provide a numerically stable way of determining where to tap this delay line, i.e., our solution does not require numerical approximations of any derivatives.

3. INTERPOLATION

We assume that the video frame rate is at least 24 fps; if it drops below 24 fps, the graphics will be noticeably jerky and we don't care anymore. This means that the Nyquist frequency of position updates is at least 12Hz, so that we can interpolate source and listener positions by applying a low-pass filter with a cut-off frequency of 12Hz or less.

In practice, a discretization of an RC low-pass filter with a cut-off frequency of 6Hz works nicely (Appendix A.2). Such a filter is defined by the equation

$$y_{i+1} = y_i + \alpha(x - y_i), \tag{3}$$

where x is the current target value (Section 4) and α is the *smoothing factor*

$$\alpha = \frac{1}{1 + \frac{f_0}{2\pi f_c}}, \tag{4}$$

where f_c is the cut-off frequency of the filter.

4. THE METHOD

Audio rendering is driven by a callback that computes one frame at a time. For each frame, we are given a buffer of new samples

[*] Current affiliation: Google Inc

emitted by the source, as well as the position of listener and source at the beginning of the frame.

Our audio renderer has a state consisting of the following pieces of information:

- a queue of sample buffers and associated source positions, waiting to be rendered,
- a low-pass filter that interpolates between positions updated at the (video) frame rate and the audio sample rate (Section 3),
- a *relative index* that measures the difference (in samples) between the current time and the beginning of the current source buffer, i.e., the buffer at the head of the queue,
- a *current index* that indicates the most recently processed sample in the current source buffer, and
- an interpolator that computes samples for fractional indices. Appendix A.3 shows a linear interpolator, chosen for brevity; more sophisticated interpolation schemes can (and should) be implemented with the same interface. Cubic interpolation is a good choice.

When computing a new output buffer, we first append the new source buffer and position to the queue, and we compute a new target position, i.e., the source position at the head of the queue relative to the current listener position. This target position will be the input value for our low-pass smoothing filter, until we advance to a new source buffer (and hence to a new source position). Then, for each output sample, we compute the next output value of our smoothing filter, obtaining an approximation of the next position vector of the source relative to the listener.

The crucial idea behind our method is the heuristic that *the length of this approximate relative position vector will be close to the exact (but computationally unattainable) distance $d(p(s), q(t))$ in Equation 1.* We simply substitute this approximation of the distance into Equation 1 and obtain an approximation of the time s at which our next sample was emitted, without having to solve any equations.

More concretely, if j is the current relative index, y is the current low-pass filter value approximating the distance between source and listener, and i is the index of the desired source sample, our heuristic yields

$$i = j - \gamma y. \tag{5}$$

If i is above the range of the current source frame, we subtract the length of the current source frame from both i and j and advance to the next source frame and position. If the index i is within the range of the current source frame, we pass i to the interpolator to obtain the the desired interpolated source sample. If i is negative, then the beginning of the current source buffer hasn't reached the listener yet, and so we only render a zero sample in this case. Finally, we increment the relative index j, advance the low-pass filter to the next value, and repeat the procedure for the next sample.

The low-pass filter updates determine how fast we step through a source buffer, and the transition to the next buffer in the queue advances the target position for the low-pass filter. In other words, low-pass filter updates and source buffers leapfrog each other; filter updates drive the progression through the current source buffer, and the progression through source buffers drives filter updates. Appendix A.1 contains a C++ implementation of our method.

f_v	24 fps	30 fps	60 fps	120 fps
q	0.50	0.57	0.74	0.86

Table 1: *Maximum admissible velocity as a fraction q of the speed of sound, for video frame rate f_v, sample rate 44100Hz, and low-pass cut-off 6Hz.*

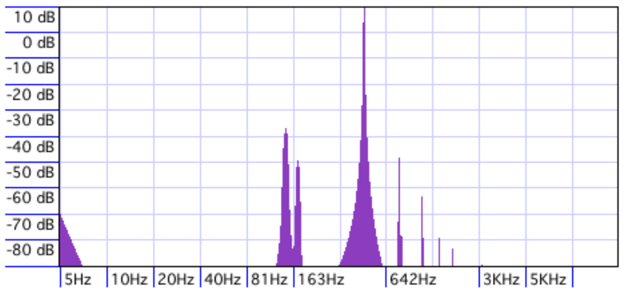

Figure 1: *Frequency spectrum of a Doppler-shifted 440Hz sine wave emitted by a source traveling at a speed of 20m/s, with a sample rate of 40960Hz, audio buffer size of 128 frames, and smoothing filter cut-off 8Hz.*

5. DISCUSSION

Figure 1 shows the frequency spectrum of a Doppler-shifted 440Hz sine wave emitted by a source traveling at a speed of 20m/s as received by a stationary listener, with a sample rate of 40960, an audio buffer size of 128 frames, and a smoothing filter cut-off frequency of 8Hz.

The main peak of the signal is at 466Hz, the desired frequency of the Doppler-shifted signal according to Equation 2. The remaining peaks are artifacts that are caused by our approximation of the Doppler shift. Note, however, that for the purposes of illustration we deliberately chose parameters that would render those artifacts visible in the spectrum. With a smaller buffer size, for example, the artefacts would be much less pronounced.

Our approach converges exponentially to the correct Doppler shift if source and listener move at constant speed relative to each other, and it produces subjectively convincing results as long as the speeds involved are not too large.

Mathematical analysis (Appendix B) yields an upper bound on admissible relative velocities of source and listener (Table 1). For typical parameters (sample rate 44100Hz, low-pass cut-off 6Hz, video frame rate $f_v = 60$ fps), this bound is roughly 74% of the speed of sound, which should be sufficient for most applications. These upper bounds on meaningful velocities are a consequence of two limitations of our approach.

One limitation is that our method does not allow for negative frequency shifts. (In theory, it is possible to hear a symphony backwards by flying away from the orchestra at twice the speed of sound.) If the listener moves at supersonic speed, our method will just fall silent because each source sample is processed exactly once and will not be revisited.

Another limitation is that due to its exponential nature, our low-pass filtering approach to position interpolation may overshoot its target when the relative speed of source and listener is around the speed of sound or larger. In particular, when the source moves away from the listener at the speed of sound, our method will not render audio at half the stationary rate (which would be the desired behavior). Instead, our source times will decrease at the beginning of each frame, causing a moment of silence, followed by a period of catching up at a rate that is slightly higher than desired.

In spite of the possibility of dropouts at high velocities, our method is robust in the sense that it will not get out of control when objects move too fast. When velocities return to the feasible range, rendering will continue normally, potentially obviating the need to check for excessive speeds.

Our approach of queuing audio input buffers implicitly creates delay line that dynamically grows or shrinks as needed. While this is often a desirable feature, it may consume too much memory when distances between sources and listeners are too large. This is easily remedied through *distance culling*: If the queue becomes too long, we cease to append new source frames and append dummy frames instead. Alternatively, one may modify our approach to employ a fixed-length delay line.

6. IMPLEMENTATIONS

The original implementation of our method is contained in the class DelayPath.java of the visualization and sonification package *jReality*[2].[1] In addition to the simulation of Doppler effects discussed here, it includes support for distance culling, auxiliary sends and returns for effects as well as location-dependent distance cues [3], and rendering backends for spatialized rendering through Ambisonics [4] or vector based amplitude panning [5].

Another implementation of our approach will be part of future versions of Csound[2] and is currently available from the Csound CVS repository. The sample code in Appendix A was adapted from the Csound version.

7. CONCLUSION

Our method of simulating sound propagation in a scene has a number of appealing features: It is fast, physically accurate and subjectively convincing as long as velocities do not get too close to the speed of sound, and robust even if velocities approach or exceed the speed of sound. Moreover, it does not require the computation of velocities or Fourier transforms, and it is amenable to a number of extensions such as position-dependent filtering, spatialization, and resampling for improved sound quality. We believe that it is a promising approach for audiovisual applications that require Doppler shifts in real time.

Acknowledgments

We are grateful to Camille Goudeseune for encouragement and a thorough critique of an early draft of this paper. Eric Lyon also provided valuable feedback.

A. CODE

The C++ code in this section is a modification of the new `doppler` opcode of Csound, currently available from the Csound CVS repository. While this implementation will compile and work as presented here (up to some boilerplate such as variable initialization), it is only intended for illustration purposes.

Any real-world implementation of our method will include additional functionality such as distance culling, support for distance cues (e.g., distance-dependent attenuation), and multi-channel encoding (e.g., stereo or Ambisonics). For a full-featured example, see the class DelayPath.java of jReality.

A.1. Main class

```
#include <cmath>
#include <list>
#include <vector>

class Doppler {

protected:
  double speedOfSound;            // meters/second
  double smoothingFilterCutoff; // Hz
  double sampleRate;              // Hz
  double samplesPerDistance;      // samples/meter
  int frameSize;  // audio frame size in samples

  RCLowpassFilter *smoothingFilter;
  LinearInterpolator *audioInterpolator;
  std::list< std::vector<double> *>
      *sourceBufferQueue;
  std::list<double> *sourcePositionQueue;
  int relativeIndex;
  int currentIndex;

public:
  void init() {
    // initialize sampleRate, frameSize,
        speedOfSound, smoothingFilterCutoff
        depending on the application

    // the remaining initializations are always
        the same
    samplesPerDistance = sampleRate /
        speedOfSound;
    audioInterpolator = new LinearInterpolator;
    smoothingFilter = 0;  // instantiate later
    sourceBufferQueue = new std::list<
        std::vector<double> *>;
    sourcePositionQueue = new std::list<double>;
    currentIndex = 0;
    relativeIndex = 0;
  }


  // the main audio processing callback
  void process(double sourcePosition, double
      micPosition, double *audioInput, double
      *audioOutput) {

    // append the new sample buffer and source
        position to the queue
    std::vector<double> *sourceBuffer = new
        std::vector<double>;
    sourceBuffer->resize(frameSize);
    for (size_t inputIndex = 0; inputIndex <
        frameSize; inputIndex++) {
      (*sourceBuffer)[inputIndex] =
          audioInput[inputIndex];
    }
    sourceBufferQueue->push_back(sourceBuffer);
    sourcePositionQueue->push_back(sourcePosition);
```

[1] `http://www.jreality.de/`
[2] `http://www.csounds.com/`

```
    // compute the new target position
    std::vector<double> *currentBuffer =
        sourceBufferQueue->front();
    double targetPosition =
        sourcePositionQueue->front() -
        micPosition;

    // initialize the smoothing filter if
        necessary
    if (!smoothingFilter) {
      smoothingFilter = new RCLowpassFilter();
      smoothingFilter->initialize(sampleRate,
          smoothingFilterCutoff, targetPosition);
    }

    for (size_t outputIndex = 0; outputIndex <
        frameSize; outputIndex++) {
      // update approximate relative position
      double relativePosition =
          smoothingFilter->update(targetPosition);
      double distance =
          std::fabs(relativePosition);

      // compute delay for relative position
      double sourceTime = relativeIndex -
          (distance * samplesPerDistance);
      int targetIndex = int(sourceTime);
      double fraction = sourceTime - targetIndex;
      relativeIndex++;

      // process samples up to target index
      while (targetIndex >= currentIndex) {
        // have we exhausted the current frame?
        if (currentIndex >= frameSize) {
          // yes: roll back indices...
          relativeIndex -= frameSize;
          currentIndex -= frameSize;
          targetIndex -= frameSize;

          // ... and advance to next frame
          delete sourceBufferQueue->front();
          sourceBufferQueue->pop_front();
          sourcePositionQueue->pop_front();
          currentBuffer =
              sourceBufferQueue->front();
          targetPosition =
              sourcePositionQueue->front() -
              micPosition;
        }
        // process current sample
        audioInterpolator->put(
            (*currentBuffer)[currentIndex++]);
      }

      // compute output sample
      double currentSample =
          audioInterpolator->get(fraction);
      audioOutput[outputIndex] = currentSample;
    }
  }
};
```

A.2. Low-pass filter

We use a discretized version of an RC low-pass filter[3] for smoothing out the difference between audio and video frame rates.

```
static double pi = std::atan(1.0) * 4.0;
```

[3] http://en.wikipedia.org/wiki/Low-pass_filter

```
class RCLowpassFilter {

public:
  void initialize(double sampleRate, double
      cutoffHz, double initialValue) {
    double tau = 1.0 / (2.0 * pi * cutoffHz);
    alpha = 1.0 / (1.0 + (tau * sampleRate));
    value = initialValue;
  }

  double update(double inputValue) {
    value += alpha * (inputValue - value);
    return value;
  }

protected:
  double alpha;
  double value;
};
```

A.3. Interpolator

A number of interpolation schemes (e.g., linear, cosine, cubic) will work in our setting. The only restriction is that we need to choose a *random access* method [6], i.e., an interpolation scheme that will give interpolated values for arbitrary fractional times. Recursive fractional delay filters will not work in this context.

```
class LinearInterpolator {

public:
  LinearInterpolator() :
    priorValue(0.0),
    currentValue(0.0) {}

  virtual void put(double inputValue) {
    priorValue = currentValue;
    currentValue = inputValue;
  }

  virtual double get(double fraction) {
    return priorValue + (fraction *
        (currentValue - priorValue));
  }

protected:
  double priorValue;
  double currentValue;
};
```

B. MATHEMATICAL ANALYSIS

We want to compute the sample heard at time $t = k$, for $k = 0, 1, 2, \ldots$ Equation 1 tells us that the sample received at time k was emitted at time

$$s_k = k - \gamma d(p(s_k), q(k)).$$

For every index k, our method yields an approximation x_k of $d(p(s_k), q(k))$, so that $s_k = k - \gamma x_k$. Similarly, we have $s_{k+1} = k + 1 - \gamma x_{k+1}$. By subtracting the first equation from the second one, we obtain

$$s_{k+1} - s_k = 1 - \gamma(x_{k+1} - x_k).$$

The distances x_k are the output of of a discretized RC low-pass filter defined by the equation $x_{k+1} - x_k = \alpha(X - x_k)$, where

$\alpha = \frac{1}{1+\tau f_0}$ and $\tau = \frac{1}{2\pi f_c}$ (Appendix A.2). X is the current target position, and f_c is the cut-off frequency of the low-pass filter. We conclude that

$$s_{k+1} - s_k = 1 - \alpha\gamma(X - x_k).$$

Our method will process each input sample exactly once, i.e., it does not revisit previously used samples. In order to avoid audio dropouts, we require the *forward propagation criterion*

$$s_{k+1} - s_k = 1 - \alpha\gamma(X - x_k) > 0.$$

This is equivalent to

$$\alpha\gamma(X - x_k) < 1.$$

Our goal is to find the largest velocity v with the property that the forward propagation criterion is satisfied for all $k \geq 0$.

Let N be the number of audio samples per video frame. Then $\Delta = \frac{Nv}{f_0}$ is the distance that an audio sample travels during one video frame.

We need to find an upper bound for $X - x_k$. We may assume that $x_0 = 0$ and $X_m = (m+1)\Delta$ is the target position after m video frames (i.e., after mN audio samples). By substituting these values into our update equation, we see that $X_0 - x_0 = \Delta$ and $X_1 - x_N = \Delta(1 + \beta^N)$, where $\beta = 1 - \alpha$ is the complement of the low-pass filter coefficient. Induction shows that $X_m - x_{mN} = \Delta(1 + \beta^N + \beta^{2N} + \cdots + \beta^{mN})$, and the geometric series tells us that

$$X - x_k < \frac{\Delta}{1 - \beta^N}$$

for all $k \geq 0$. Hence, the forward propagation criterion is satisfied if

$$\frac{\alpha\gamma\Delta}{1 - \beta^N} < 1.$$

Substituting $\Delta = \frac{Nv}{f_0}$ into this equation yields the desired upper bound on our velocity:

$$\frac{v}{c} < \frac{1 - \beta^N}{N\alpha}.$$

Table 1 shows typical upper bounds for $q = \frac{v}{c}$ computed according to this formula. Note that for constant speeds v, the lag between our target position X and our current position x_k converges exponentially to $\frac{\Delta}{1-\beta^N}$, so that on average, for constant speed, we have exponential convergence to the correct frequency shift.

C. REFERENCES

[1] F. R. Moore, *Elements of Computer Music*, Prentice-Hall, Inc., 1990.

[2] Steffen Weiß mann, Charles Gunn, Peter Brinkmann, Tim Hoffmann, and Ulrich Pinkall, “jReality: a java library for real-time interactive 3D graphics and audio,” in *Proceedings of the 17th ACM international conference on Multimedia*, New York, NY, USA, 2009, MM ’09, pp. 927–928, ACM.

[3] Richard W.E. Furse, “Spatialisation - Stereo and Ambisonic,” in *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing,and Programming*, 2000.

[4] Michael Gerzon, “Surround sound psychoacoustics,” 1974.

[5] Ville Pulkki, “Spatial sound generation and perception by amplitude panning techniques,” Tech. Rep., Helsinki University of Technology, 2001.

[6] Julius O. Smith, *Physical Audio Signal Processing, December 2008 Edition*, `http://ccrma.stanford.edu/~jos/pasp/`, 2008, online book.