Implementing Andy Farnell's 'bouncing ball' in Csound

Marijana Janevska, James Anderson, Joachim Heintz

Incontri – Institute for contemporary music at the HMTM Hannover

janevskam@stud.hmtm-hannover.de

Abstract: This paper discusses the Csound implementation of the bouncing ball model from Andy Farnell's *Designing Sound* [Farnell 2010]. We will consider Farnell's approach to sound design with Pure Data, then present two possible procedures for extending this model to and improving it in Csound. Finally, we will present creative examples of varying and employing the model in a musical context.

Keywords: Csound, CsoundQt, Pure Data, Hannover, Incontri, FMSBW, Andy Farnell, Designing Sound

1 Introduction

During the last two semesters' programming seminar at the Hochschule für Musik, Theater und Medien Hannover's Incontri Institute for New Music, we have focused on transferring sound models in Andy Farnell's *Designing Sound* from Pure Data to Csound, one of which was the bouncing ball. In this paper we will follow Andy Farnell's description of a bouncing ball and its acoustical characteristics. How can a bouncing ball be depicted? What model can be developed to replicate its sound? How is this model implemented in PD and Csound? How can it be applied creatively to a musical context?

2 Analysis following Andy Farnell's description of the model of a bouncing ball

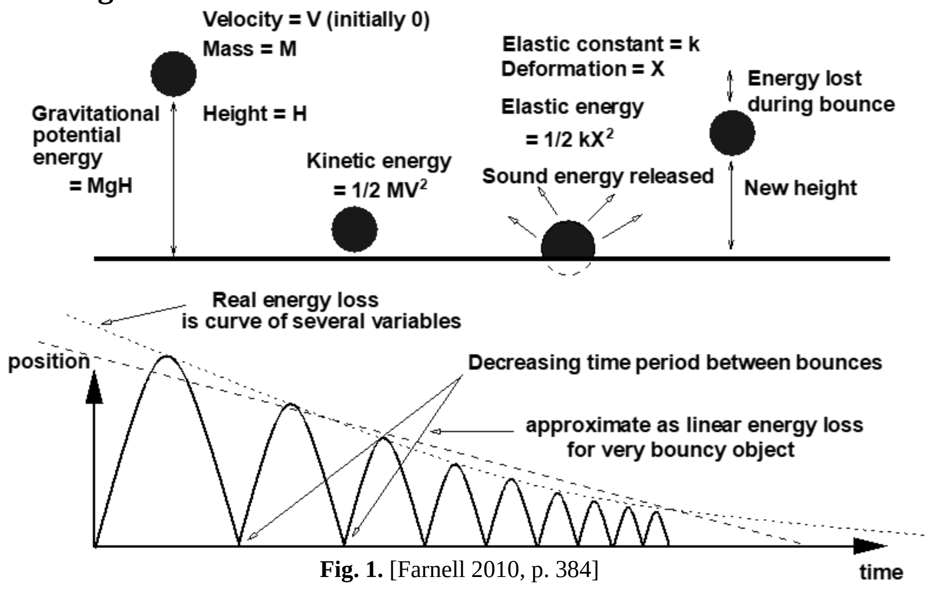
Velocity = V (initially 0)
Mass = M
Height = H
Gravitational potential energy = MgH
Kinetic energy = 1/2 MV^2
Elastic constant = k
Deformation = X
Elastic energy = 1/2 kX^2
Sound energy released
Energy lost during bounce
New height
Real energy loss is curve of several variables
position
Decreasing time period between bounces
approximate as linear energy loss for very bouncy object
time

Fig. 1. [Farnell 2010, p. 384]

A bouncing object’s physical behaviour is characterized as positional energy lost over time. Farnell’s analysis is focused on the decaying energy during a sequence of events. A pattern is used to represent the sequence in which the duration between events as well as the spectral complexity and total amplitude in each event diminish. Of course, the size, density, material content and initial height of the ball can all affect the impact sound, which can vary enormously. The rate of energy loss can be approximated as linear. Since the energy of each impact is a result of the height from which the ball falls, the first bounce is the loudest and longest in duration. It also deforms the ball the most, and results in a much richer sound compared to the subsequent bounces. The first bounce would have the most complex harmonic spectrum, whereas the ensuing bounces’ spectra would linearly become simpler.

3 Program flow as general model

The basis for each implementation is the development of a general model that can be implemented in any relevant programming language. Thus, following Farnell’s description of the model, we produced the following program diagrams and then used it to transfer the model to Csound:

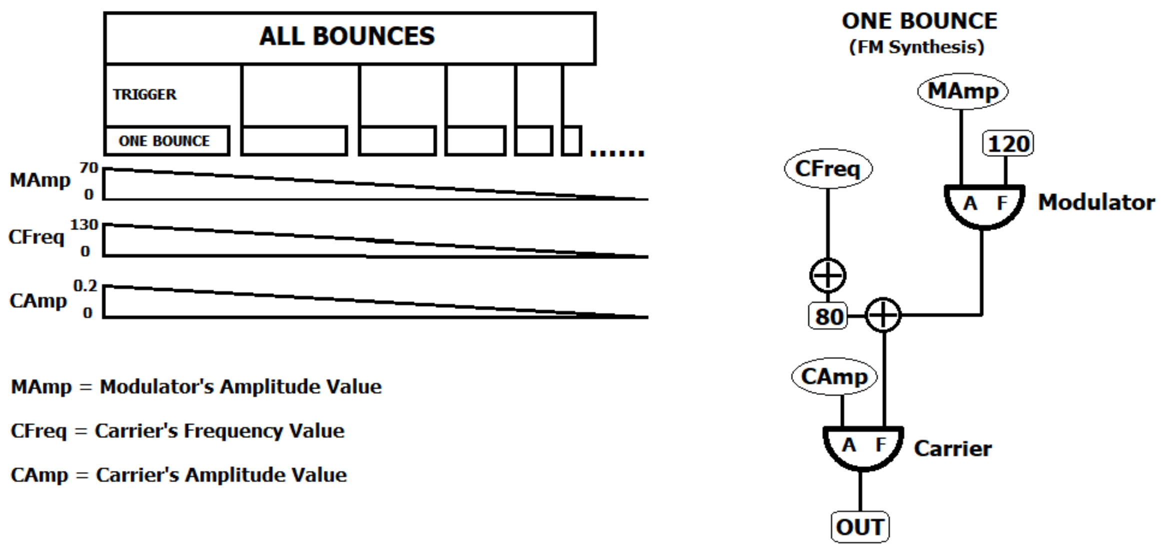

Fig. 2. Program diagrams

The two diagrams illustrate two important aspects of the model.

- Firstly, the ALL BOUNCES unit produces the temporal sequence. Each bounce acts as a trigger to the ONE BOUNCE unit, with the time between each decreasing as the sequence progresses. The ALL BOUNCES unit also creates the decaying lines for some of the variables used in the ONE BOUNCE units, namely Modulation Amplitude, Carrier Frequency and Carrier Amplitude.

- Secondly, the ONE BOUNCE unit produces the sound for each individual bounce. This is achieved via frequency modulation, which uses the variables generated by the ALL BOUNCES unit as well as locally generated envelopes.

4 Farnell's process for implementing the model in PD

There are many possible ways to implement the above model, all of which will vary depending on the program used. Farnell's PD implementation consists of two parts. In the first, for generating one bounce, frequency modulation synthesis is used to reduce the spectral complexity of each bounce. This is repeatedly triggered by a metronome in the second part of the model, and is responsible for the sound of each bounce.

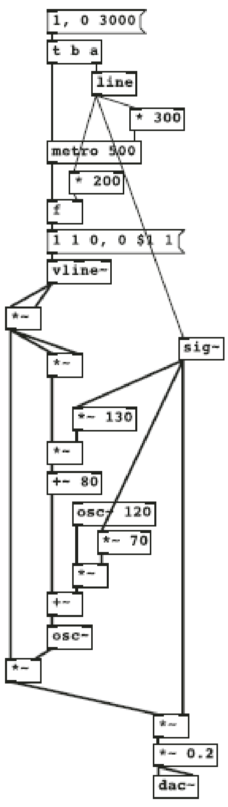

Fig. 3. [Farnell 2010, p. 385]

A linear envelope from 1 to 0 over a 3 second period is first constructed for the overall duration of the sequence of events. A bang message starts a metronome (built with the 'metro' object) that triggers each individual event. The time between the first and second bounces (the triggering time) is 0.3 seconds, and gradually falls to 0 over the entire sequence of bounces, resulting in ever more frequent triggers. The initial event's decay time is 0.2 seconds, which also drops to 0 over the course of the sequence, and is stored in the 'float' object. A new value is consequently sent to the 'vline' object with each trigger, which produces a linear ramp that progresses from 0 to 1 in 1 millisecond, and moves from 1 to 0 over the decay time sent from the 'float' object. This then adjusts the level of frequency modulation and the main output amplitude for each subsequent event, starting with an amplitude of 0.2 for the first bounce and falling to 0 at the end of the event-sequence.
The frequency modulation synthesis, used for producing the sound of each bounce, begins with a complex sound that is rich in harmonics, and gradually reduces it to a sine wave. More precisely, the frequency of the carrier starts at 210Hz and falls to 80Hz; the amplitude of the modulator begins at 70 for the first bounce, then progresses to 0. The modulator frequency is 120 Hz, which is a constant value.

5 Implementing the model in Csound

We have considered two of many possible ways for implementing the model in Csound:

a. Straightforwardly and in some ways similar to the PD version, we built two instruments: First, the "One_bounce" instrument produces the sound of each bounce using frequency modulation; the "All_bounces" instrument then uses the 'metro' opcode to trigger the "One_bounce" instrument with a decreasing time period between subsequent triggers.

```
instr All_bounces
 kLine = linseg:k(1, p3, 0)
 kMetroFreq = divz(1,kLine*0.3,1000)
 kTrigger = metro(kMetroFreq)
  if  kTrigger == 1 then
   schedulek("One_bounce",0,kLine*0.2,kLine)
  endif
endin

instr One_bounce
 iDecay = p4 //receives what is kLine in 'All_bounces'
 aMod = poscil:a(iDecay*70,120)
 aEnv = linseg:a(0,0.002,1,p3-0.002,0)^2
 aCarFreq = 80+130*iDecay*aEnv^2
 aCar = poscil:a(iDecay*0.2,aCarFreq+aMod)
 outall(aCar*aEnv)
endin

schedule("All_bounces",0,3)
```

Though this implementation remains relatively faithful to the PD version, there are nevertheless a number of noteworthy differences. By separating the ALL BOUNCES and ONE BOUNCE units, Csound's instrument approach arguably better represents the model than the PD version. Moreover, the variable names present the opportunity to express in the code what takes place in the programme. Two details should be mentioned here. Since the 'metro' opcode in Csound has a frequency input argument (rather than PD's 'metro' object which requires a duration as an input), the metro frequency must be calculated as the inverse of the kLine signal. To avoid the possible division by zero in writing 1/kLine, we instead use the 'divz' opcode. In the PD version, Farnell avoids clicks when a new bounce is triggered by using "1 1 0, 0 $1 1" as input for the 'vline~' object, rather than the simpler "1, 0 $1 1". The envelope in Csound's "One_bounce" instrument could be employed without a fade-in, e.g.:

```
aEnv = linseg:a(1,p3,0) ^ 2
```

We nevertheless decided to include a short fade-in of two milliseconds in order to achieve a less noisy attack. This is one example of "fine tuning" the sound, for which Csound offers a wealth of possibilities.

b. Csound also presents the possibility of an instrument that triggers itself, so that the model can be implemented with one self-triggering instrument that stops when a given limit is reached, in this case 1/1000 of a second.

```
instr Bouncing_ball
 iThisDuration = p3
 iDecay = (3-p2)/3
 aMod = poscil:a(iDecay*70,120)
 aEnv = linseg:a(0,0.002,1,p3-0.002,0)^2
 aCarrFreq = 80+130*iDecay*aEnv^2
 aCar = poscil:a(iDecay*0.2,aCarrFreq+aMod)
 outall(aCar*aEnv)
 if p3>1/1000 then
   schedule("Bouncing_ball",iThisDuration*1.5, iThisDuration*0.9)
 endif
endin
schedule("Bouncing_ball",0,.2)
```

6 Expanding the model in Csound[1]

There are many possible ways of expanding the model and employing it creatively in a musical setting. We will present just a few of these using the first method of implementing the model:
- Introducing p-fields to change the relevant parameters in the instrument
- Or let the parameters be chosen in a random range
- Since the modulator frequency is the only constant value in the model, adding a moving envelope will lead to a wide variety of frequency modulation values and a very rich palette of sonic possibilities
- Using very long events, or reversing the progression of the event durations from short to long
- Overlapping events over time

```
instr All_bounces
 kLine = linseg:k(p4, p3, p5)
 kMetroFreq = divz(1,kLine*0.3,1000)
 kTrigger = metro(kMetroFreq)
  if  kTrigger == 1 then
   schedulek("One_bounce",0,kLine*0.2,kLine)
  endif
endin

instr One_bounce
 iDecay = p4 //receives what is kLine in 'All_bounces'
 kMFRandom = random:k(1, 1000)
 kMFRandom2 = random:k(1/10, 10)
 kModFreq = randomi:k(p5, p5+kMFRandom, kMFRandom2)
 aMod = poscil:a(iDecay*70,kModFreq)
 aEnv = linseg:a(0,0.002,1,p3-0.002,0)^2
 aCarFreq = 1+p6*iDecay*aEnv^2
 aCar = poscil:a(iDecay*0.2,aCarFreq+aMod)
 outall(aCar*aEnv*0.2)
endin

</CsInstruments>
<CsScore>
;                 s.t.   dur.  s.L.  e.L.  1bDur  MF    CF
i "All_bounces" 0      6.7    1     0      4     930   4400
i "All_bounces" 6.7    7.3    0     1      1.8   760   250
i "All_bounces" 12.1   9.2    0     1      3.5   718   47

</CsScore>
</CsoundSynthesizer>
```

[1] All the examples can be found at: https://github.com/incontri-hannover/ICSC2022/tree/main/Examples

7 References

1. Andy Farnell: Designing Sound, Cambridge: MIT Press (2010)
2.Examples of composed etudes can be found at:
https://github.com/incontri-hannover/ICSC2022/tree/main/Examples