An Euterpea Widget for customized instruments

CPSC 432 Final Project

December 18, 2009

- Creating custom instrument in Euterpea: type Instr a = Dur -> AbsPitch -> Volume -> [Double] -> a

- Issues to deal with when creating instruments: clipping, enveloping, rendering time, tedious testing

- InstrVisualizer widget: quickly switch between instruments, fine tuning parameters, preview wave form without loading an external app (eg. Audacity), save to file, play, etc

- A good way to test the combination of 2 separate components of Euterpea: Signal/UI used in GMI and signal functions implemented as Arrow.

Introduction

Euterpea, a music library written in Haskell, does not only provide a concise way to write music melody with great expressiveness, but also add great functionalities to manipulate such melodies. Euterpea offers a powerful way to program the "interpretation" of musical values. Euterpea can play a piece of music in Midi, using whichever instrument specified in the melody. In addition, we can also create out own instrument and have Euterpea play that instrument or export out to a wav file. The process of designing complex models of instruments is completely separated from the music melody itself.

Like everything else in Euterpea, we first look at the type. In this case, the type of an instrument is:

```
type Instr a = Dur -> AbsPitch -> Volume -> [Double] -> a
```

Note that its type is polymorphic so as to create either Stereo and Mono sounds determined by *signal function a*

Many instruments can be transcribed over to Euterpea from CSound files. In addition, the programmer has the freedom to tweak the parameters to create different sound that suits her taste or need. There should be a quick way of testing the sound output after any change to the instrument. This project proposes InstrVislizr which is aimed at easing that process.

Meet InstrVislizr

We will start with a simple instrument, improve it along the way with InsVislizr.

```
f1 = gen10 4096 [1]
simple :: Instr (Mono AudRate)
simple dur pch vol pfields =
   proc _ do
     oscil f1 - <- apToHz pch
```

This plays a sine wave of the frequency given by a given note. It is rather monotonous. We would like to improve it by adding some extra effects on it. For example, we use enveloping method to make the sound more interesting. With some experience, we will come up with something like this:

```
simple :: Instr (Mono AudRate)
simple dur pch vol pfields =
   let envSF = linseg [1,0.99,0.4,0.2,0,0] (map f durations)
       f d = d / sum durations * fromRational dur
       durations = [0.02, 0.38, 0.4, 0.2, 0]
   in proc _ -> do
      s <- oscil f1 0 -< apToHz pch
      amp <- envSF -< ()
      returnA -< s * amp * 0.2
```

It is apparent that there are a lot of numerical parameters in the example above. Normally, the process of obtaining "good values" for a good sound would involve the following steps:

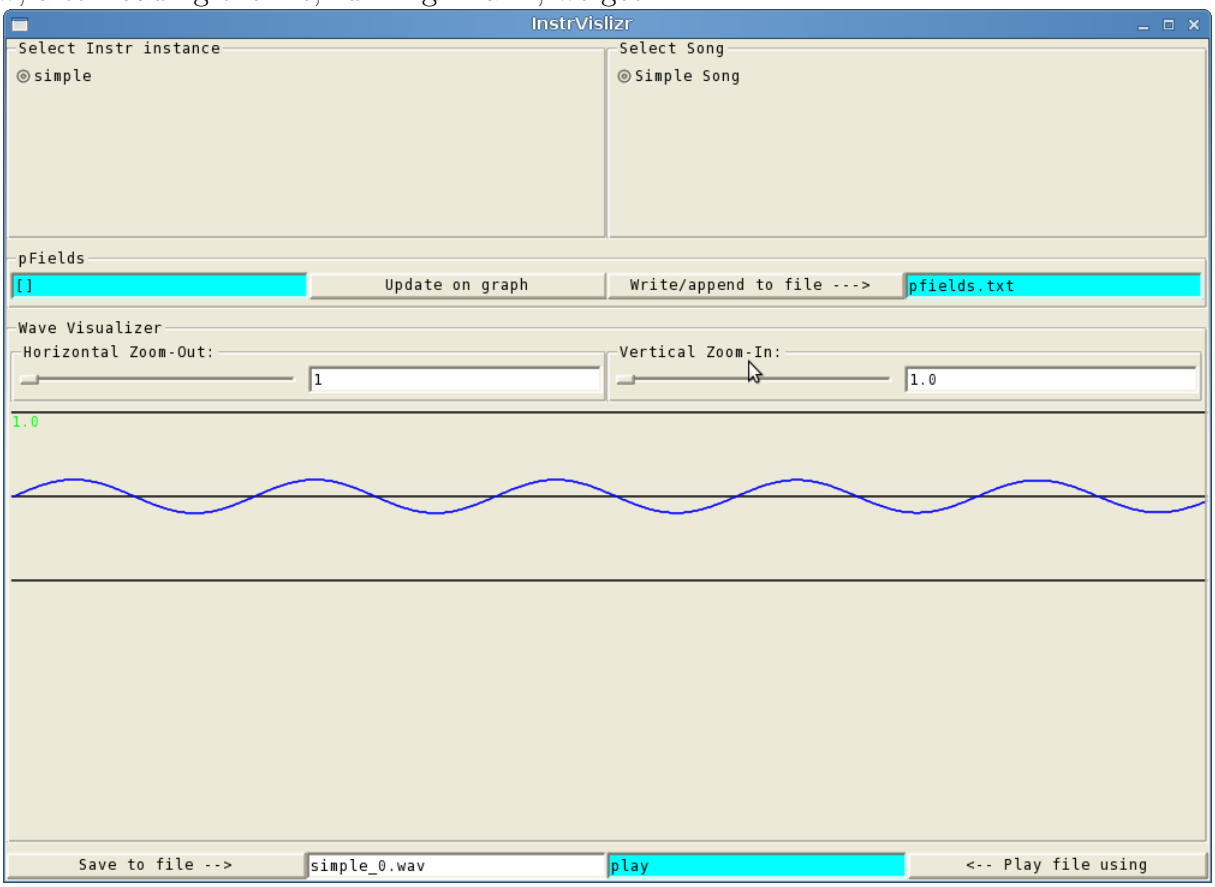
1. Punch in a number of notes and numbers
2. Render out to a wav file
3. Run a third party app and open the wave file to view and play
4. Have an idea of some areas to tweak, occasionally just intending to modify some numerical parameters
5. Go back to Haskell text editor and change the numbers
6. Repeat the above steps

That is so tedious! Is there a better way?

Meet InstrVislizr.

```
import InstrVislizr
-- CODE THAT YOU ALREADY HAVE ----------------
...
simple = ... -- as shown above
simpleInsr = Custom ‘‘simple’’ :: InstrumentName
instrMap :: InstrMap (Mono AudRate)
instrMap = [(Custom ‘‘simple’’, simple)]
simpleSong = a 4 qn
-----------------------------------------------
-- CODE NEEDED FOR InstrVislizr --------------
songMap = [(‘‘Simple Song’’, simpleSong)]
main = vislizr instrMap songMap []
-- THAT’S IT ---------------------------------
```

Now, after loading the file, running “main”, we get



Things you can do with InstrVislizr:

1. Choose any instrument from the list
2. Choose any song from the list

3. Immediately see waveform output created by the picked instrument playing the picked song
4. Waveform is plotted before clipping so it is easier to examine that Audacity when a curve is clipped
5. Zoom in/zoom out quickly
6. Change pFields values and if applicable to that instrument, see changes to the waveform by clicking "Update on graph" (more on pFields below)
7. Store any good pFields value for future use
8. Save the sound of the respective waveform to a wave file
9. Play the sound from a wave file

The above snippet of code shows the use of *vislizr*, a function that takes an instrument map, a song map and an optional pfields map (pass [] otherwise)

```
vislizr :: (Euterpea.Audio.Types.AudioSample a, Euterpea.Audio.Types.Clock p) =>
        Euterpea.Audio.Render.InstrMap (Euterpea.Audio.Types.Signal p () a)   ->
        [(String, Euterpea.Music.Music Euterpea.Music.Pitch)]                 ->
        [(Euterpea.Music.InstrumentName, [Double])]   ->
        IO ()
```

It is polymorphic to support both InstrMap (Mono AudRate) and InstrMap (Stereo AudRate) instances. AudRate can be also exchanged with CtrRate.

The power of PFields

We rewrite the *simple* function above as follow:

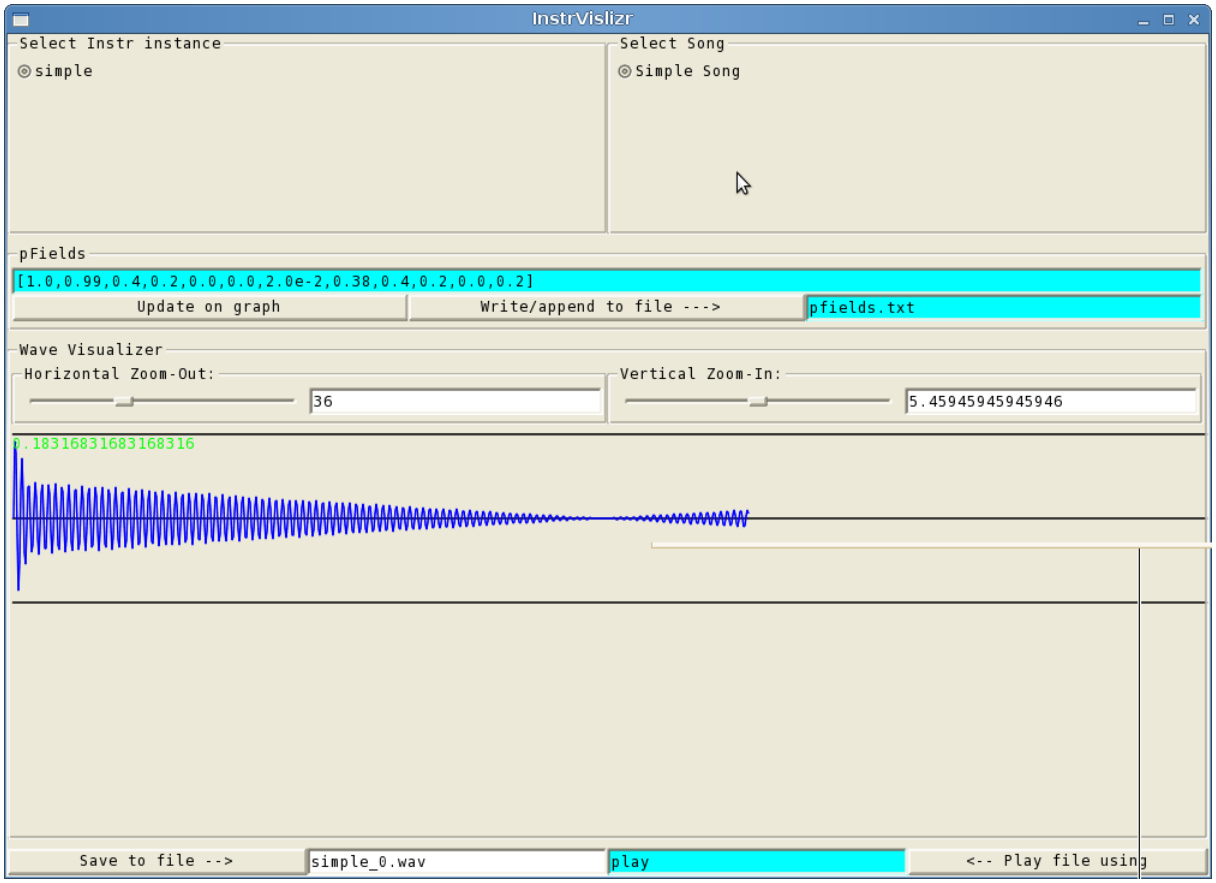
```
simple :: Instr (Mono AudRate)
simple dur pch vol pFields | length pFields > 10 =
  let amps = take 5 pFields
      durations = take 5 (drop 5 pFields)
      f d = d / sum durations * fromRational dur
  in proc _ -> do
     s <- oscil f1 0 -< apToHz pch
     amp <- linseg amps (map f durations) -< ()
     returnA -< s * amp * 0.2


simple dur pch vol _ | otherwise = -- handle default case
     simple dur pch vol [1,0.99,0.4,0.2,0,0, -- first 5 values (amp)
                         0.02, 0.38, 0.4, 0.2, 0, -- next 5 values (dur)
                         0.2] -- final scale factor
```

It is clear that we can now overwrite various parameters that affect the sound wave OUTSIDE the signal function *simple* itself.

The real advantage kicks in when pFields values can be changed on-the-fly in InstrVislizr.

```
simpleInstr = Custom "simple"
instrMap = [(simpleInstr, simple)]
songMap = [("Simple Song", simpleSong)]
pFieldsMap = [(simpleInstr, [1,0.99,0.4,0.2,0,0, 0.02, 0.38, 0.4, 0.2, 0, 0.2])]
main = vislizr instrMap songMap pFieldsMap
```



`pFields can also be used in other situations other than for enveloping functions.`

Can InstrVislizr really help?

In short, yes. Try changing the pFields with other numerical values, see the new graph, click "Write/append to file" (the cyan textboxes are editable) and all the values are kept in a file for later uses, especially easy copy and paste in the format of a Haskell list (e.g. to replace the previous set of default pFields values)

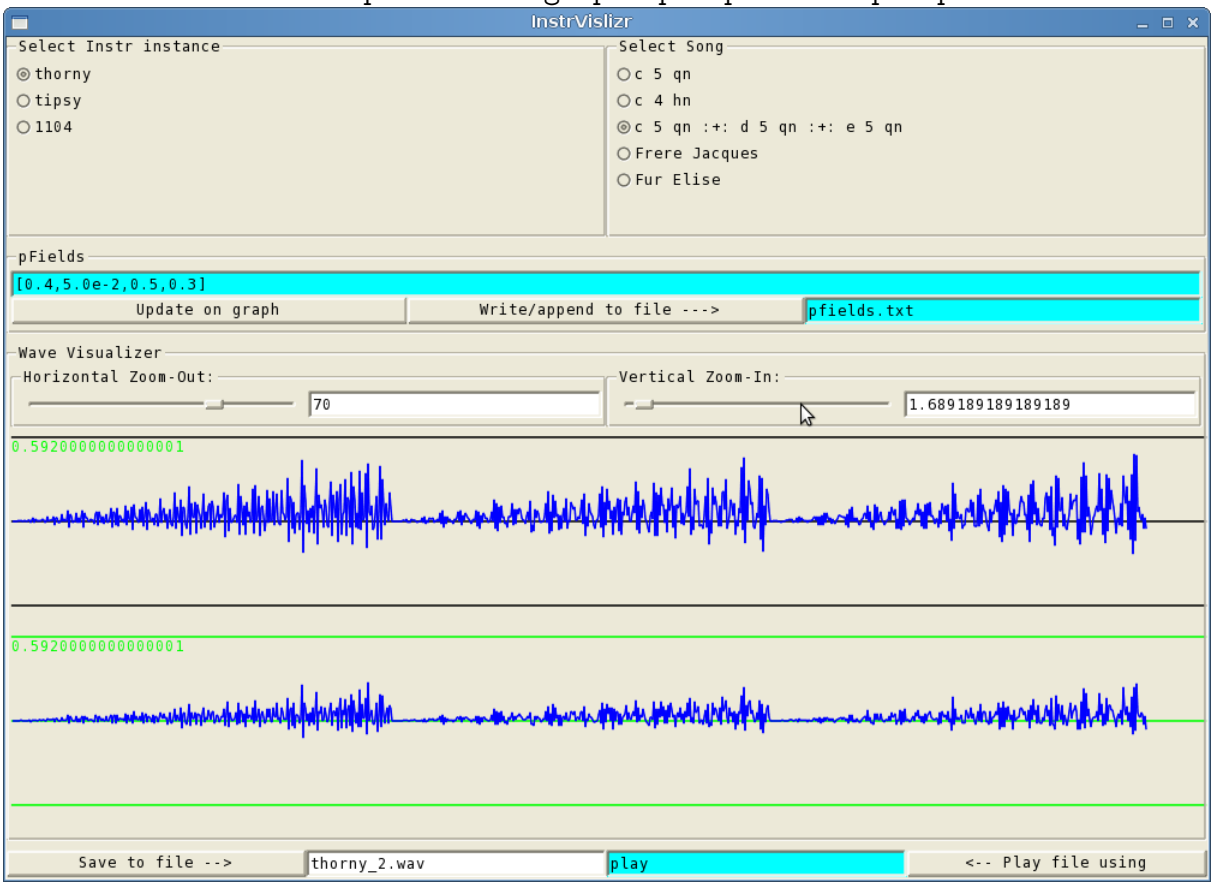
Besides the *vislizr* function, InstrVislizr module provides another version that allows customizing the dimension of the graphical window (everything else will shrink proportionally)

```
vislizrEx ::
  (AudioSample a, Clock p, Show a1) =>
  (Int, Int) -- FOR WIDTH AND HEIGHT
  -> InstrMap (Signal p () a)
  -> [(String, Music Pitch)]
  -> [(InstrumentName, [a1])]
  -> IO ()
```

To get one started quickly, InstrStore.hs provides some examples of instrument code and pFields valuesk, and ScoreStore.hs contain some examples of melodies. The examples are

meant to ilustrate InstrVislizr's usefulness, not to demo how great each instrument is. Therefore the minimum code to get InstrVislizr up and running is

```
import InstrVislizr
import InstrStore
import ScoreStore
main = vislizr instrMapStereo songMapSample pFieldsMapSample
```



Technical implementations

Textbox widget

Many thanks go to Reynard Le who wrote the original textbox widget in Fall 2008. It supported text input and allows initialization with a set string. The textbox widget had gone through some drastic changes listed below

- Dynamically updates text content from external source (e.g. use Signal String instead of Signal). This is very useful because a textbox value can now be updated by a slider as well as keyboard input

- Simplified some of the code structure by eliminating the spanS, spanE elements. I use fewer state variables, but added a variable to store the previous value to detect *Signal String* changes
- Optimized performance by traversing the list only once to update the redrawing instead of twice (when the cursor position is in the middle of the text)
- Added the blinking cursor/caret for easier navigation (this feature was intended but did not work in the original version)
- Added quick jumping to beginning and end of text by UP and DOWN arrow keys
- Changed background color to differentiate against Display widget :)

ForkIO

Because "Save to file" may take a long time, the IO operation is used with forkIO function. However this does not work well in ghci in general, especially in Windows and Mac. It works best with running compiled code instead. Because of this glitch, I commented out forkIO functionality – this will prevent the interactivity during a long IO operation such as Save to file and Play [music] file buttons at the bottom

IO monad vs UI monad

The UI monad contains Graphic and Sound.

```
newtype Graphic = Graphic (IO ()) -- SOE.lhs line 87
type Sound = IO ()                -- UIMonad.lhs line 204
```

Some IO () operations were required in this library, and although it does not make much sense semantically, saving file (which writes to disk) or playing file (which execute a sytem command) was handled though the Sound state. I provided a useful help that helps "lift" an IO () to UI ()

```
io2ui2 :: (a -> IO ()) -> Signal a -> UI ()
io2ui2 ioAction stuff = UI aux
  where aux ctx inp = (out <*> stuff <*> inp, (nullLayout, ()))
        out = pure (\x (ievt,s) -> ((nullGraphic, ioAction x),s))
```

Because both Graphic and Sound are restricted to IO (), it does not, however, seems possible to execute an IO a (not IO ()) operation within a UI monad.

For more interesting aspects of the code, such as graph drawing, please take a glance at InstrVislizr.hs

Conclusion

InstrVislizr aims at making it easier to visualize the sound product created by instruments implemented in Haskell, help a composer multitask and improve his productivity. Implemented with Euterpea's limited Graphic Music Interface, InstrVislizr also helps push for the development of more basic widgets that would be useful for Euterpea users.

To a small extent, InstrVislizr connects Euterpea's two separate components that are not playing nice with each other yet, the UI Monad and audio Signal Functional interface. The graph is drawn lazily depending on how many points are asked to plot on the screen, which InstrVislizr would extract from the sound signal function. It gets slower as the number of plottable samples become larger (by zooming out further). If UI Monad and Signal Functional can eventually talk to each other with respect to their infinite stream structure, many improvements can be made on the graph, including animation and user interactivity.

There are still many limitations with InstrVislizr, such as not being able to do sound playback from memory, buggy (segfault) threading issues, but I believe this is a contribution towards physical modelling of sound instruments in Euterpea.