For string orchestra

Hans Höglund 2012

January 12, 2012

Introduction

```
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}

module Music.Projects.MusicaVitae
where

import Music.Utilities
import Music.Model.Temporal.Media

import Temporal.Music.Notation hiding (delay)

import qualified Temporal.Music.Notation.Note as Note
import qualified Temporal.Music.Notation.Scales as Scales
import qualified Temporal.Music.Notation.Demo as Demo
import qualified Temporal.Music.Notation.Demo.GeneralMidi as Midi
```

Instrumentation and tuning

The instrumentation is as follows:

- Violin I-IV
- Viola I-II
- Cello I-II
- Double Bass

A basic idea of the piece is to combine (slightly) different tunings of the instruments using open-string techniques and harmonics. For this purpose, we will split the ensemble into three sections, each using a different tuning:

- Odd-numbered Vl, Vla and Vc parts tunes A4 to 443 Hz (A3 to 221.5 Hz)
- Even-numbered Vl, Vla and Vc parts tunes A4 to 437 Hz (A3 to 218.5 Hz)
- Double bass tunes A1 to 55 Hz

The other strings should be tuned in relation to the A-string as usual.

To represent this in Haskell, we must first define the data types to represent parts, sections and tunings:

```
data Part
    = Violin Int
    | Viola Int
    | Cello Int
    | DoubleBass
    deriving ( Eq, Show )

data Section
    = High | Low | Middle
    deriving ( Eq, Show )

type Tuning = Double
```

We then define the relation between these types as follows:

```
partSection   :: Part -> Section
sectionTuning :: Section -> Tuning
partTuning    :: Part -> Tuning

partSection (Violin 1) = High
partSection (Violin 2) = Low
partSection (Violin 3) = High
partSection (Violin 4) = Low
partSection (Viola 1)  = High
partSection (Viola 2)  = Low
partSection (Cello 1)  = High
partSection (Cello 2)  = Low
partSection DoubleBass = Middle

sectionTuning Low    = 434
sectionTuning Middle = 440
sectionTuning High   = 446

partTuning = sectionTuning . partSection
```

Then add some utility definitions to quickly access the various parts:

```
ensemble                                        :: [Part]
sectionParts                                    :: Section -> [Part]

isViolin, isViola, isCello                      :: Part -> Bool

highParts, lowParts                             :: [Part]
highViolinParts, highViolaParts, highCelloParts :: [Part]
lowViolinParts, lowViolaParts, lowCelloParts    :: [Part]

ensemble
    = [ Violin 1, Violin 2, Violin 3, Violin 4
      , Viola 1, Viola 2, Cello 1, Cello 2, DoubleBass ]

sectionParts s = filter (\x -> partSection x == s) ensemble

highParts = sectionParts High
lowParts  = sectionParts High

isViolin (Violin _) = True
isViolin _          = False
isViola  (Viola _)  = True
isViola  _          = False
isCello  (Cello _)  = True
isCello  _          = False
```

```
highViolinParts = filter isViolin (sectionParts High)
highViolaParts  = filter isViola  (sectionParts High)
highCelloParts  = filter isCello  (sectionParts High)
lowViolinParts  = filter isViolin (sectionParts Low)
lowViolaParts   = filter isViola  (sectionParts Low)
lowCelloParts   = filter isCello  (sectionParts Low)
```

All parts may be doubled. If several parts are doubled but not all, the musicians should strive for a balance between the two main tuning sections (i.e. avoid doubling just the upper parts or vice versa).

Certain cues are required to be played by a single musician even if the parts are doubled, which will be marked *solo*. These passages should be distributed evenly among the musicians, instead of being played by designated soloists.

```
data Doubling = Solo | Tutti
    deriving ( Eq, Show )
```

Pitch

Dynamics

```
data Dynamics = PPP | PP | P | MP | MF | F | FF | FFF
    deriving ( Show,
               Eq,
               Enum,
               Bounded )

instance Seg Dynamics

instance Vol Dynamics where
    volume = Volume (1e-5, 1)

-- short-cuts

ppp’, pp’, p’, mp’, mf’, f’, ff’, fff’ :: LevelFunctor a => a -> a
```

```
ppp’ = setLevel PPP
pp’  = setLevel PP
p’   = setLevel P
mp’  = setLevel MP
mf’  = setLevel MF
f’   = setLevel F
ff’  = setLevel FF
fff’ = setLevel FFF

dim :: LevelFunctor a => Accent -> Score a -> Score a
dim v = dynamics ((-v) *)

cresc :: LevelFunctor a => Accent -> Score a -> Score a
cresc v = dynamics (v * )
```

Playing techniques

The piece makes use of different playing techniques in both hands. As the intonation will be different between open and stopped strings, we also define a function mapping each left-hand technique to a stopping.

```
data Str
    = I
    | II
    | III
    | IV
    deriving ( Eq, Show )

data Stopping
    = Open
    | QuarterStopped
    | Stopped
    deriving ( Eq, Show )

data Articulation = Articulation
    deriving ( Eq, Show )
```

```
data Phrasing
    = Phrasing { attackVel  :: Double
               , sustainVel :: [Double]
               , releaseVel :: Double
               , staccatto  :: Double }
    deriving ( Eq, Show )

data RightHand a
    = Pizz   a Articulation
    | Single a Articulation
    | Phrase [a] Phrasing
    | Jete   [a] Phrasing
    deriving ( Eq, Show )

data LeftHand
    -- Open string techniques
    = OpenString Str
    | NaturalHarmonic Int Str
    | NaturalHarmonicTrem Int Int Str
    | NaturalHarmonicGliss Int Int Str
    | QuarterStoppedString Str
    | StoppedString Int Str
    | StoppedStringTrem Int Int Str
    | StoppedStringGliss Int Int Str
    deriving ( Eq, Show )

type Technique = RightHand LeftHand

data Cue
    = Cue { cuePart      :: Part,
            cueDoubling  :: Doubling,
            cueTechnique :: Technique }
    deriving ( Eq, Show )
```

```
class Stopped a where
    stopping :: a -> Stopping

instance Stopped LeftHand where
    stopping ( OpenString           _     ) = Open
    stopping ( NaturalHarmonic      _ _   ) = Open
    stopping ( NaturalHarmonicTrem  _ _ _ ) = Open
    stopping ( NaturalHarmonicGliss _ _ _ ) = Open
    stopping ( QuarterStoppedString _     ) = QuarterStopped
    stopping ( StoppedString        _ _   ) = Stopped
    stopping ( StoppedStringTrem    _ _ _ ) = Stopped
    stopping ( StoppedStringGliss   _ _ _ ) = Stopped

instance Stopped a => Stopped (RightHand a) where
    stopping ( Pizz x _ ) = stopping x
    stopping ( Single x _ ) = stopping x
    stopping ( Phrase (x:xs) _ ) = stopping x
    stopping ( Jete   (x:xs) _ ) = stopping x
```

Intonation

Many playing techiniques in the score calls for open strings. In this case intonation is determined solely by the tuning.

In some cases, open-string techniques are used with an above first-position stop. This should make the open string pitch rise about a quarter-tone step (or at least less than a half-tone step).

Where stopped strings are used, intonation is determined by context:

- In solo passages, intonation is individual. No attempt should be made to synchronize intontation (on long notes et al) for overlapping solo cues.
- In unison passages, common intonation should be used.

```
data Intonation
    = Tuning
    | Raised
    | Common
    | Individual
    deriving ( Eq, Show )

intonation :: Doubling -> Technique -> Intonation

intonation Tutti t = case stopping t of
    Open           -> Tuning
    QuarterStopped -> Raised
    Stopped        -> Common

intonation Solo t = case stopping t of
    Open           -> Tuning
    QuarterStopped -> Raised
    Stopped        -> Individual
```

Rendering

```
instance Seg Int where

renderCue :: Dur -> Cue -> Score (Note.Note Dynamics Int ())
renderCue dur (Cue part doubl tech) =
    case tech of
        Pizz x attr ->
            note dur $ Note.Note (volume $ level MF)
                       (Pitch scale (tone 60))
                       Nothing

        Single x attr ->
            note dur $ Note.Note (volume $ level MF)
                       (Pitch scale (tone $ pitch x))
                       Nothing

        Phrase xs attr ->
            note dur $ Note.Note (volume $ level MF)
                       (Pitch scale (tone 60))
                       Nothing
        Jete xs attr ->
            note dur $ Note.Note (volume $ level MF)
                       (Pitch scale (tone 60))
                       Nothing

    where tune = partTuning part
          scale = makeScale tune
          intone = intonation doubl tech
          pitch (OpenString str) = undefined
          pitch (NaturalHarmonic n str) = undefined
          pitch (NaturalHarmonicTrem m n str) = undefined
          pitch (NaturalHarmonicGliss m n str) = undefined
          pitch (QuarterStoppedString str) = undefined
          pitch (StoppedString p str) = p
          pitch (StoppedStringTrem p q str) = undefined
          pitch (StoppedStringGliss p q str) = undefined
          makeScale = Scales.eqt 69

renderCuesToMidi :: Score Cue -> Score Demo.MidiEvent
renderCuesToMidi = Midi.marimba . dfoldMap renderCue

exportCues :: FilePath -> Score Cue -> IO ()
exportCues path = Demo.exportMidi path . renderCuesToMidi

play score = do
    exportCues "test.mid" score
    openMidiFile "test.mid"

export score = do
    exportCues "test.mid" score
    exportMidiFile "test.mid"
```

Form

```
t1 p = note (1/8) $ Cue (Violin 1) Solo (Single (StoppedString p I) Articulation)
t2 p = note (1/8) $ Cue (Violin 2) Solo (Single (StoppedString p I) Articulation)


test =  loop 500 t1’
        |||
        (stretch 1.008 $ loop 500 t1’)
    where
        t1’ = t1 60 >>> t1 65 >>> t1 67 >>> t1 60 >>> t1 55 >>> t1 62

scale = line [ note (1/4) $ Cue (Violin 1)
                            Solo (Single (StoppedString p I) Articulation)
             | p <- [60..72] ]
```