For string orchestra

Hans Höglund 2012

January 11, 2012

Introduction

This file contains the code for a (yet to be named) piece for string orchestra. It is a literate source file, meaning it is written both for reading and running as a program. Throughout, code will appear in a `typewriter font`. The code form a valid program that can be run to produce the piece.

If you obtain this document in source form, you can convert it to a PDF document using Pandoc [1] or compile it using a Haskell compiler[2].

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

module Music.Projects.MusicaVitae
where

import Data.Monoid
import Data.Foldable

import Music.Utilities

import Temporal.Music.Notation hiding (tmap, dmap, tdmap)
```

[1] Which can be found at http://johnmacfarlane.net/pandoc

[2] Such as GHC, see http://www.haskell.org/ghc

```
import qualified Temporal.Media as Media
import qualified Temporal.Music.Notation.Note as Note
import qualified Temporal.Music.Notation.Pitch as Pitch
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    = 437
sectionTuning Middle = 440
sectionTuning High   = 443

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)
```

Musical preliminaries

We are going to represent time and pitch using two packages for Haskell called *temporal-media* and *temporal-music-notation* [3].

[3] All packages we reference here can be obtained from http://hackage.haskell.org

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 LeftHand
    -- Open string techniques
    = OpenString Str
    | NaturalHarmonic Int Str
    | NaturalHarmonicTrem Int Int Str
    | NaturalHarmonicGliss Int Int Str

    -- Quarter stopped string techniques
    | QuarterStoppedString Str

    -- Stopped string techniques
    | StoppedString Int Str
    | StoppedStringTrem Int Int Str
    | StoppedStringGliss Int Int Str
    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


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

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

type Technique = RightHand LeftHand

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

data Cue
    = Cue { cuePart      :: Part
```

```
        , cueDoubling  :: Doubling
        , cueTechnique :: Technique }
    deriving (Eq, Show)
```

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 | stopping t == Open           = Tuning
                    | stopping t == QuarterStopped = Raised
                    | stopping t == Stopped        = Common
intonation Solo   t | stopping t == Open           = Tuning
                    | stopping t == QuarterStopped = Raised
                    | stopping t == Stopped        = Individual
```

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

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

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

Form

```
test1 = note 1 $ Cue (Violin 1) Solo (Pizz $ OpenString I)
test2 = test1 +:+ test1 +:+ test1 +:+ test1
```

Rendering

```
instance Seg Int where
```

```
-- Add instances to fold a score of scores

class DurFunctor f where
    dmap :: (Dur -> a -> b) -> f a -> f b

class DurFoldable t where
    foldDmap :: Monoid m => (Dur -> a -> m) -> t a -> m

instance Monoid (Media.Media Dur a) where
    mempty  = rest 0
    mappend = (+:+)

instance DurFunctor (Media.Media Dur) where
    dmap = Media.dmap

instance Foldable (Media.Media Dur) where
    foldMap render = mconcat . events . renderScore . fmap render
        where events (EventList _ xs) = map eventContent xs

instance DurFoldable (Media.Media Dur) where
    foldDmap render = mconcat . events . renderScore . dmap render
        where events (EventList _ xs) = map eventContent xs


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

        Single x ->
            note dur $ Note.Note (volume $ level MF)
```

```
                        (Pitch scale (tone 60))
                        Nothing

        Phrase (x:xs) attrs ->
            note dur $ Note.Note (volume $ level MF)
                        (Pitch scale (tone 60))
                        Nothing

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

    where tune = partTuning part
          scale = makeScale tune
          intone = intonation doubl tech


makeScale :: Tuning -> Pitch.Scale
makeScale = Scales.eqt 69

renderCuesToMidi :: Score Cue -> Score Demo.MidiEvent
renderCuesToMidi = Midi.stringEnsemble1 . foldDmap renderCue

exportCues :: Score Cue -> IO ()
exportCues = Demo.exportMidi "test.mid" . renderCuesToMidi

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

openMidiFile = openFileWith "/Applications/Utilities/QuickTime Player 7.app/Contents/MacOS/Q
```

```
exportMidiFile = openFileWith "/Applications/Sibelius 6.app/Contents/MacOS/Sibelius 6"
```