Exploring Melody Space in a Live Context Using Declarative Functional Programming

FARM Workshop at ICFP 2014, Gothenburg

Thomas G. Kristensen, uSwitch Ltd, London

Composer is a simple, responsive and extensible system utilising logic programming to allow novices to explore and learn music rules

Background

Background

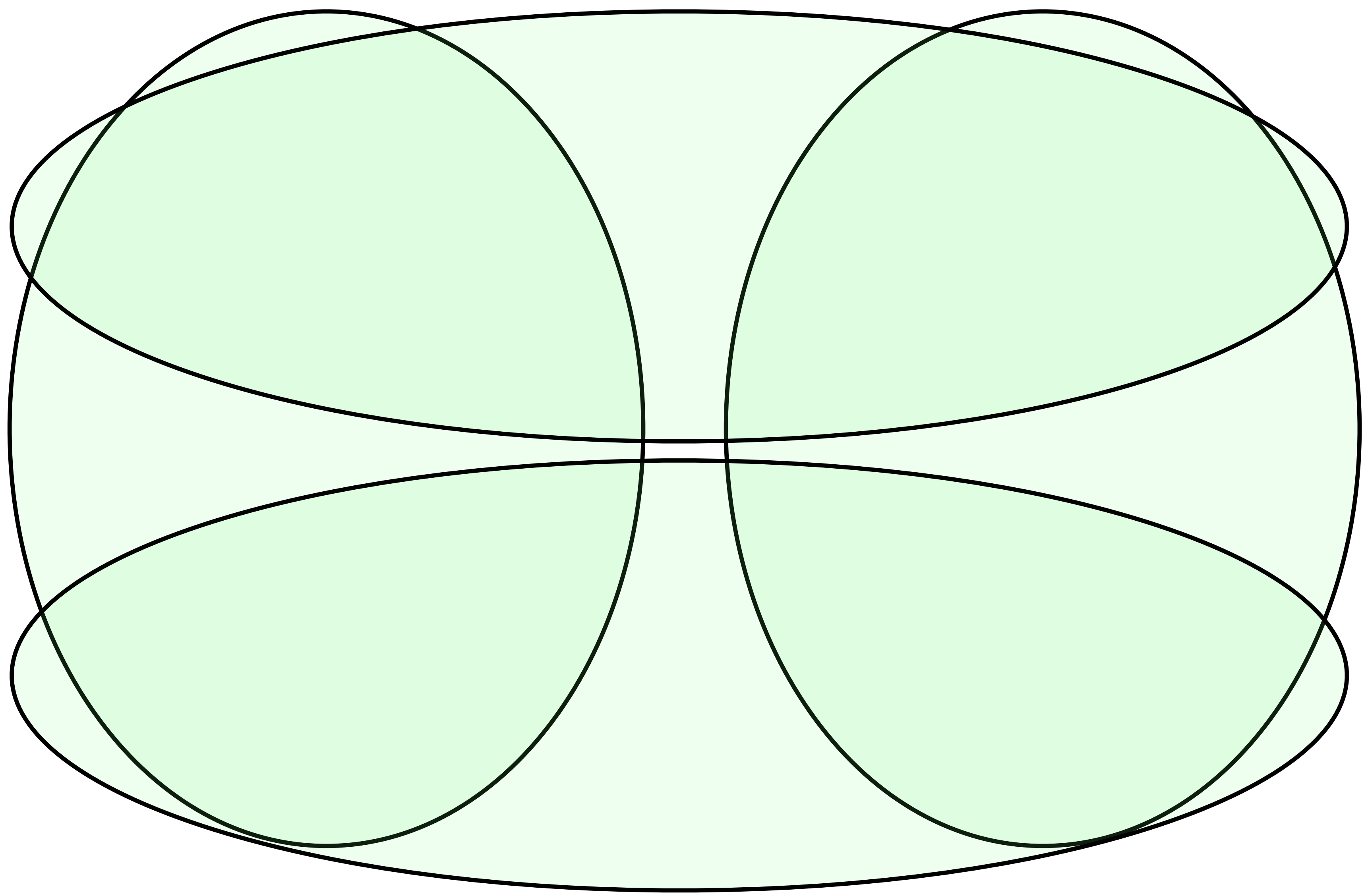
offline

online

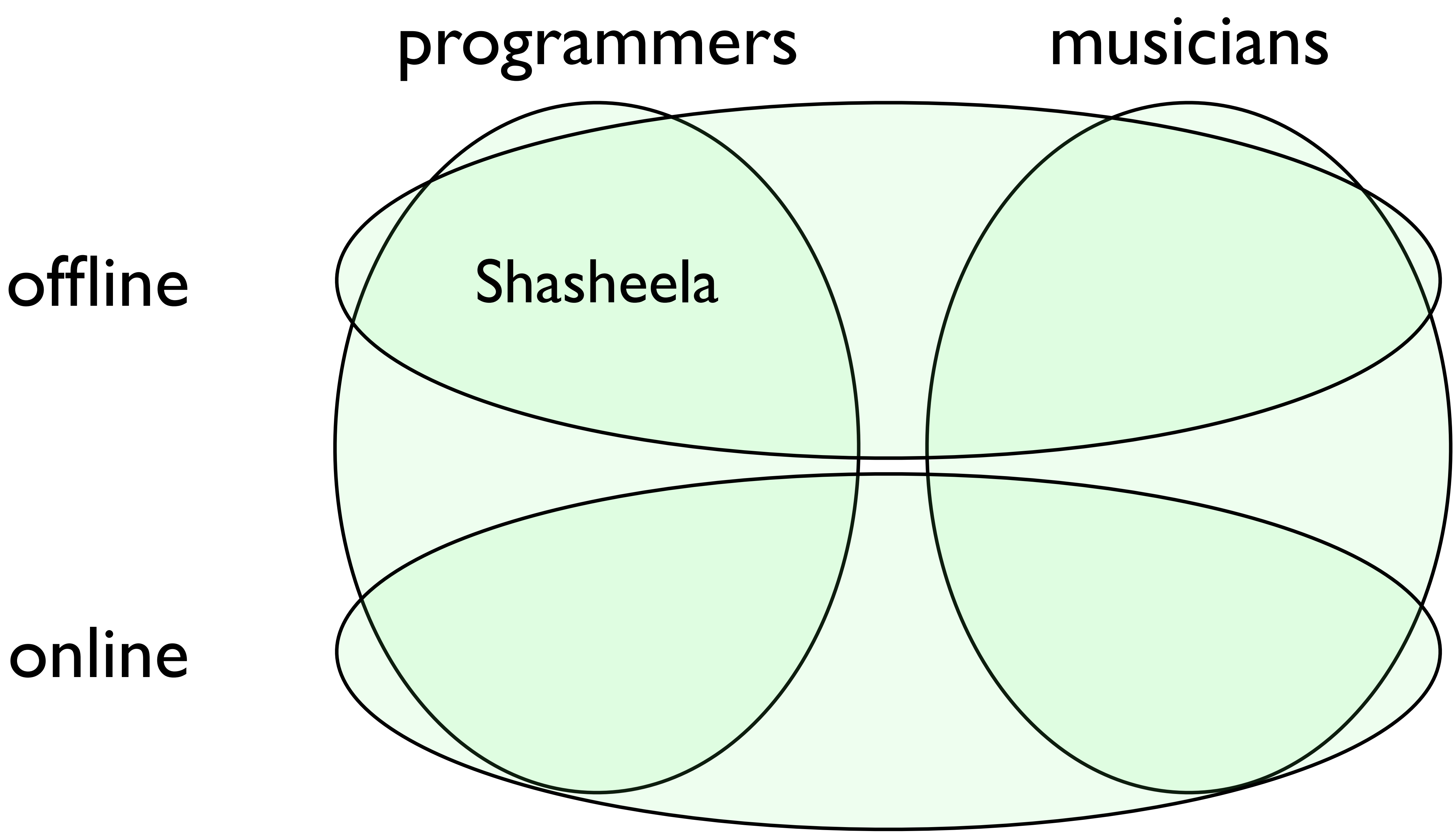
Background

	programmers	musicians
offline		
online		

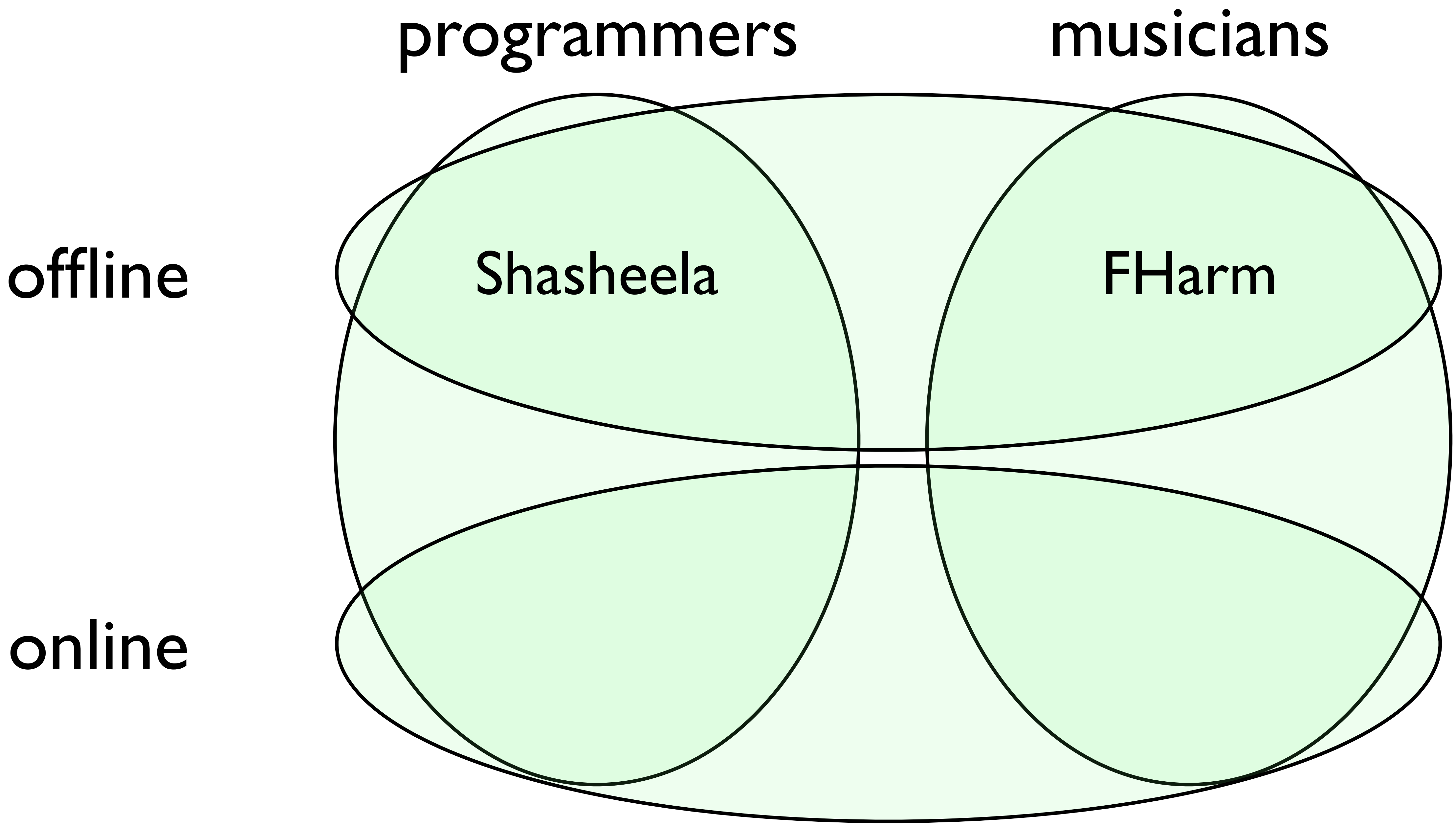
Background



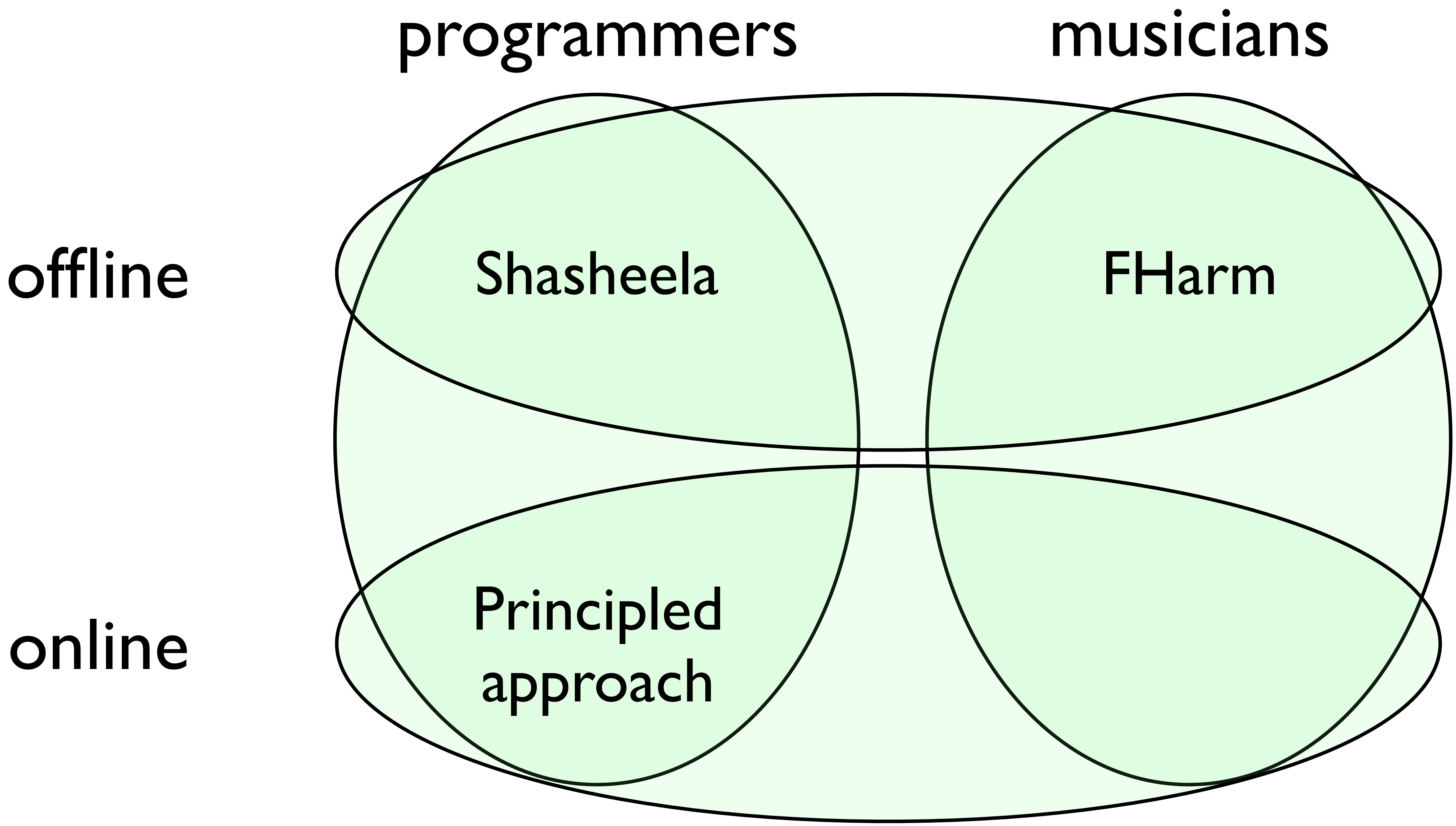
Background



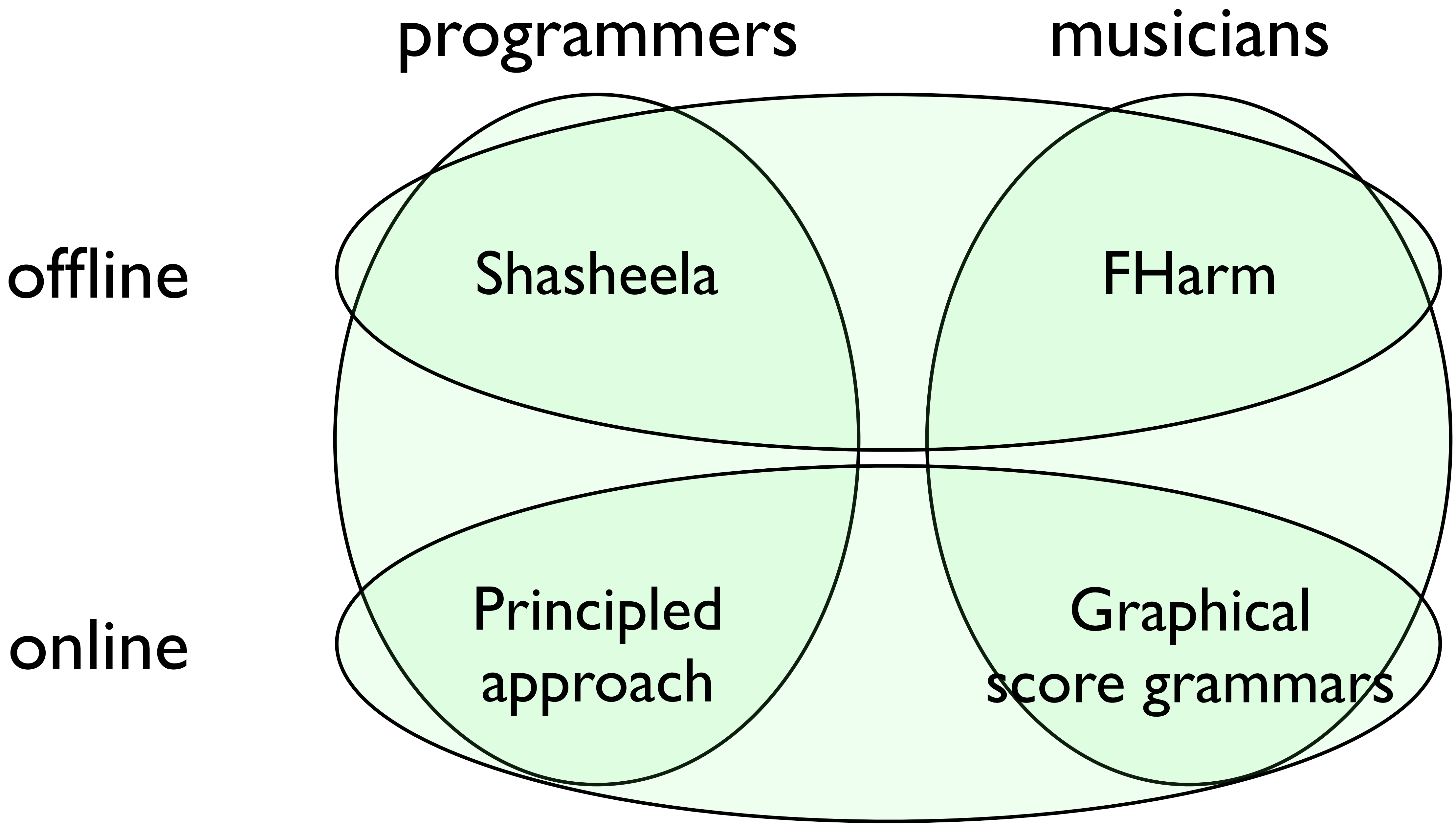
Background



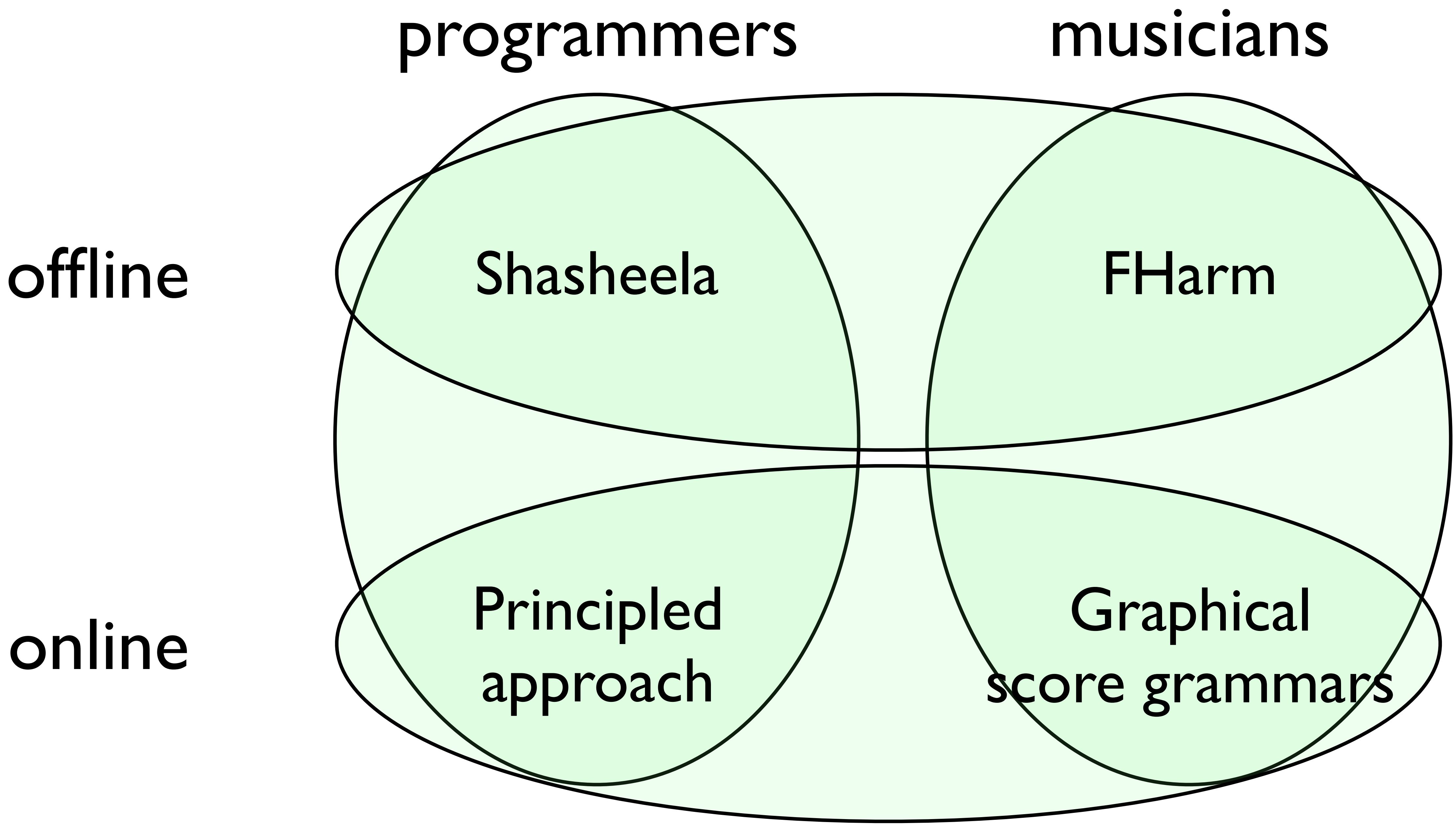
Background


programmers
musicians
offline
online
Shasheela
FHarm
Principled
approach

Background


programmers
musicians
offline
online
Shasheela
FHarm
Principled
approach
Graphical
score grammars

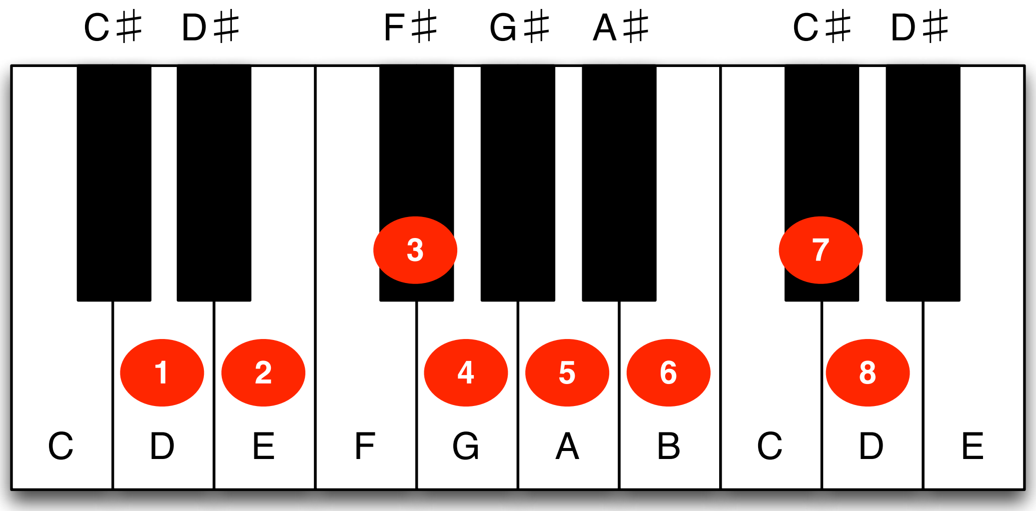
Background



- Anders: Composing music by composing rules (Ph.D. thesis)
- Koops, Magalhãe and de Haas: A functional approach to automatic melody harmonisation
- Aaron, Blackwell, Hoadley and Regan: A principled approach to developing new languages for live coding
- Stead, Blackwell and Aarong: Graphic score grammars for end-users

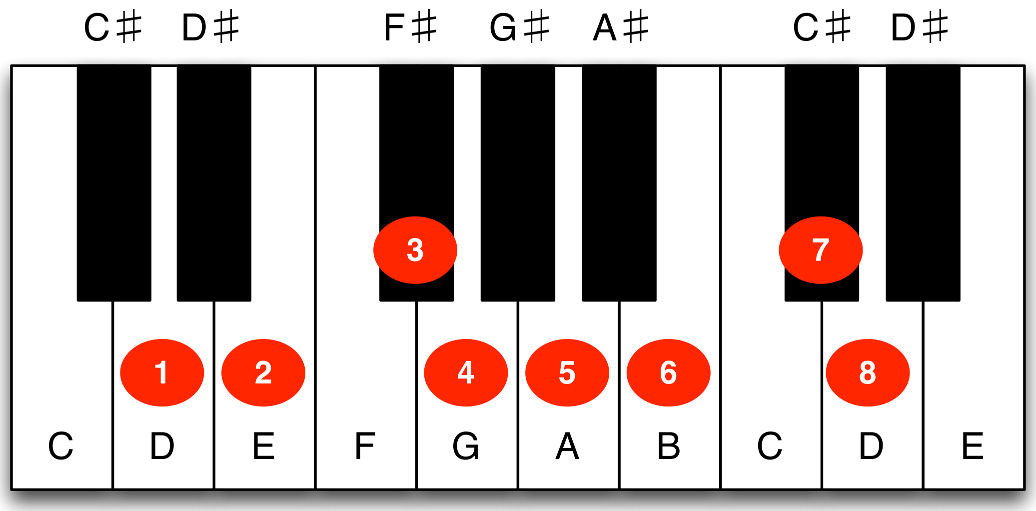
Melody rules

- Tonic note
- Mode
- Cadence

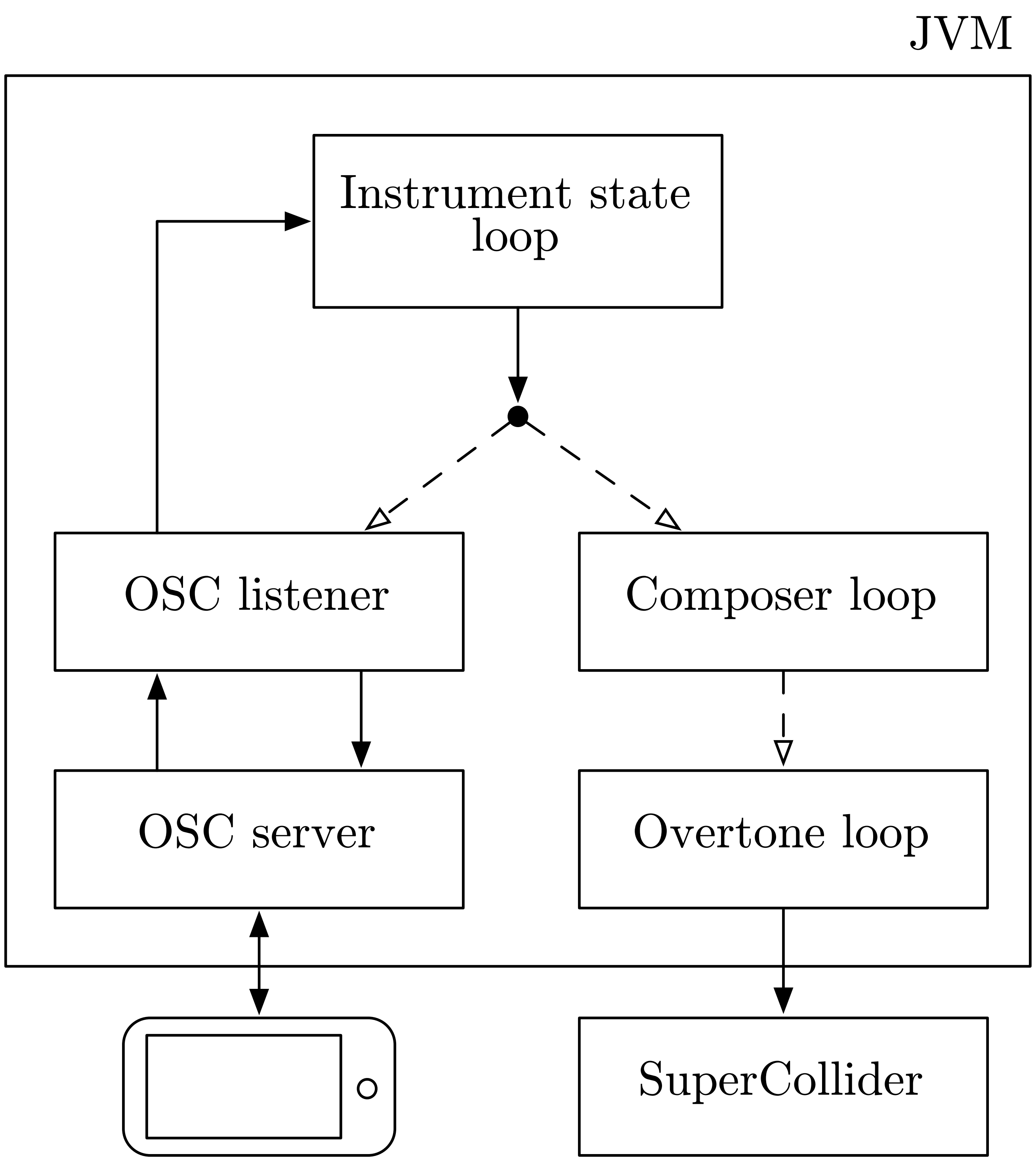


Melody rules

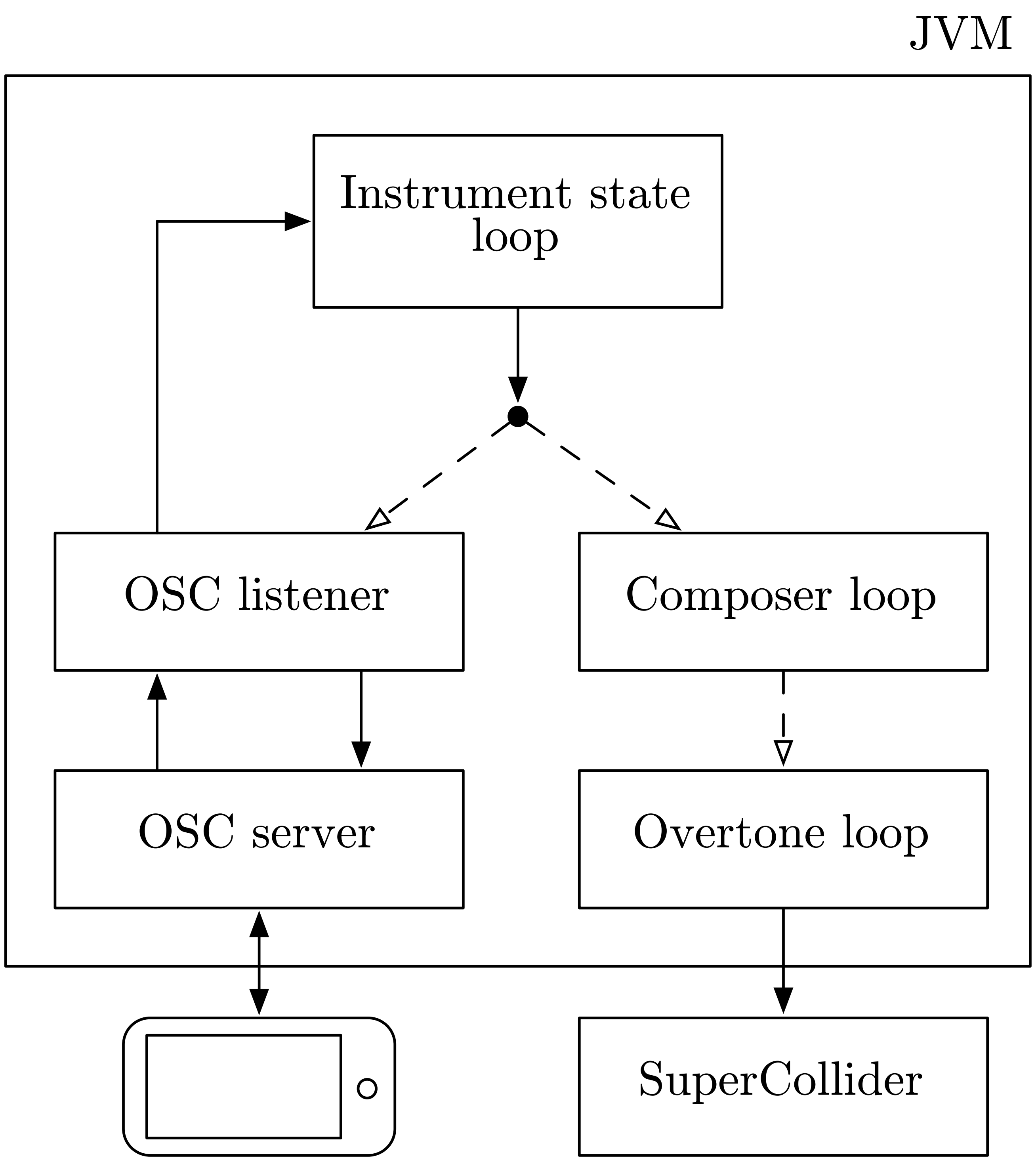
- Tonic note
- Mode
- Cadence

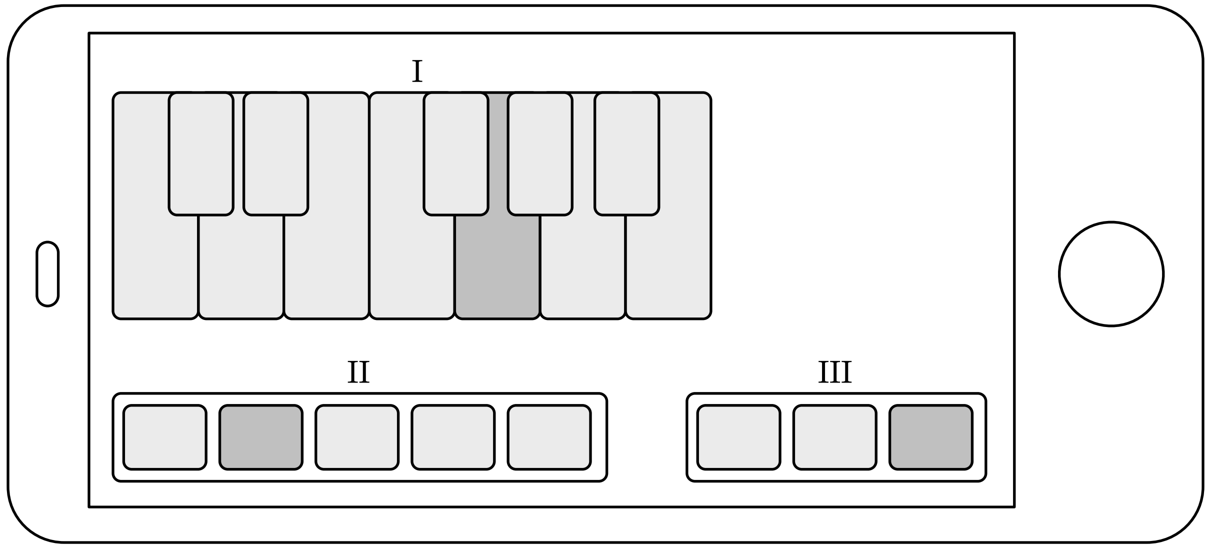


Architecture



Architecture





Logic programming

Logic programming

```
(run* [notes]
      (scaleo :C3 major-scale notes)
      (counto notes 8))
;; => ([:C3 :D3 :E3 :F3 :G3 :A3 :B3 :C4])
```

Logic programming

```
(run* [notes]
      (scaleo :C3 major-scale notes)
      (counto notes 8))
;; => ([:C3 :D3 :E3 :F3 :G3 :A3 :B3 :C4])
```

```
(run 3 [m1 m2 m3 m4 m5 m6 m7 m8]
     (fresh [n1 n2 n3 n4 n5 n6 n7 n8]
            (scaleo :C3 major-scale
                    [n1 n2 n3 n4 n5 n6 n7 n8])
            (permuteo [m1 m2 m3 m4 m5 m6 m7 m8]
                      [n1 n2 n3 n4 n5 n6 n7 n8])
            (== m1 :C3)
            (== m8 :C4)))
;; => ([:C3 :D3 :E3 :F3 :G3 :A3 :B3 :C4]
;;     [:C3 :E3 :D3 :F3 :G3 :A3 :B3 :C4]
;;     [:C3 :F3 :D3 :E3 :G3 :A3 :B3 :C4])
```

Logic programming

```
(run* [notes]
      (scaleo :C3 major-scale notes)
      (counto notes 8))
;; => ([:C3 :D3 :E3 :F3 :G3 :A3 :B3 :C4])
```

```
(run 3 [m1 m2 m3 m4 m5 m6 m7 m8]
     (fresh [n1 n2 n3 n4 n5 n6 n7 n8]
            (scaleo :C3 major-scale
                    [n1 n2 n3 n4 n5 n6 n7 n8])
            (permuteo [m1 m2 m3 m4 m5 m6 m7 m8]
                      [n1 n2 n3 n4 n5 n6 n7 n8])
            (== m1 :C3)
            (== m8 :C4)))
;; => ([:C3 :D3 :E3 :F3 :G3 :A3 :B3 :C4]
;;     [:C3 :E3 :D3 :F3 :G3 :A3 :B3 :C4]
;;     [:C3 :F3 :D3 :E3 :G3 :A3 :B3 :C4])
```

```
(run* [tonic-note pattern]
      (scaleo tonic-note pattern
              [:C3 :D3 :E3 :F3 :G3 :A3 :B3 :C4]))
;; => ([:C3 (1 0 1 0 1 1 0 1 0 1 0 1 1 . _0)])
```

Logic programming

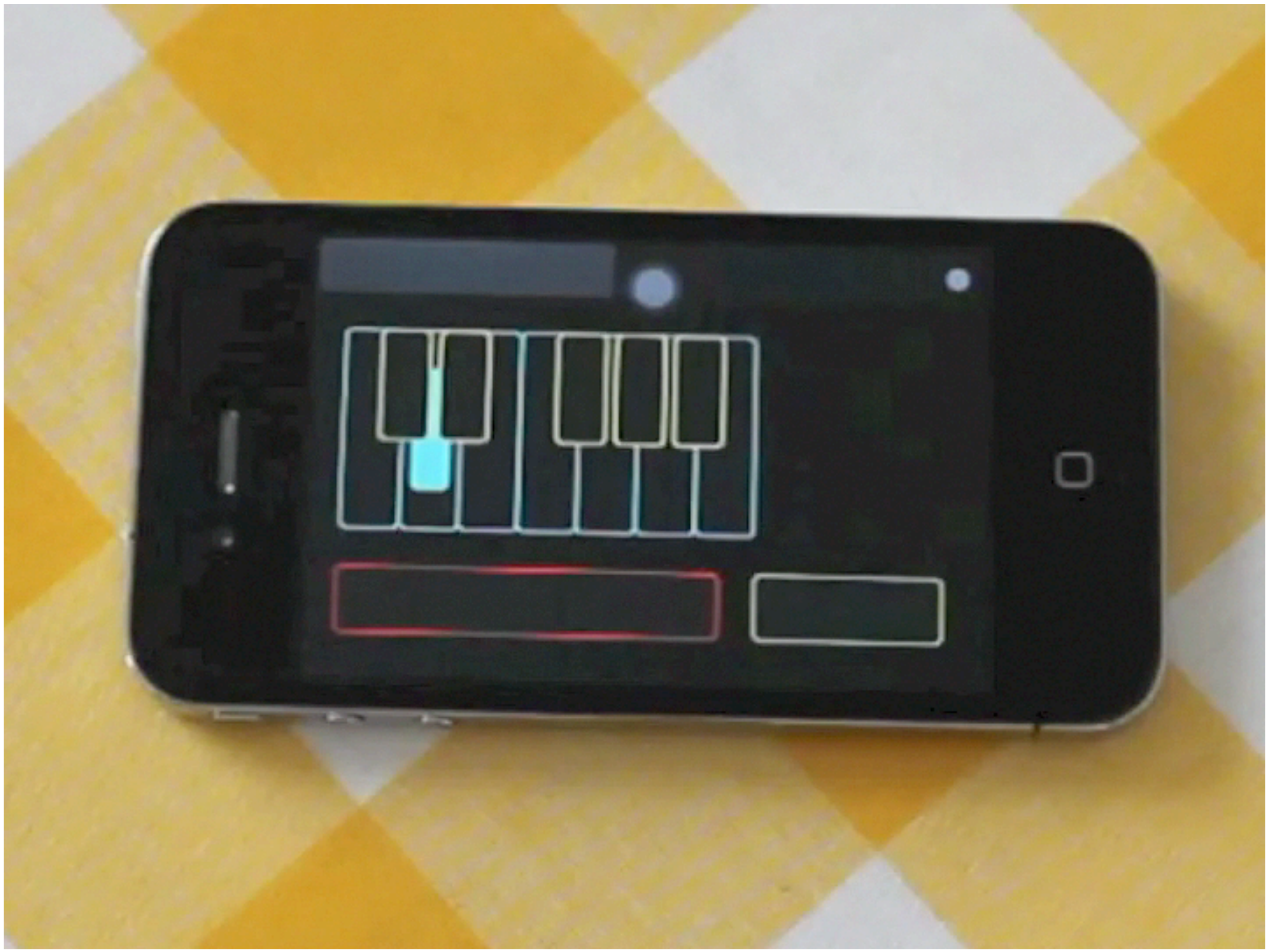
```clojure
(ns composer.composer
  (:refer-clojure :exclude [==])
  (:require [clojure.core.async :refer [go >! <!]]
            [clojure.core.logic :refer :all]
            [clojure.core.logic.pldb :refer :all]))

(defn scale-from-tones [tone-types]
  (take 25
        (->> tone-types
             (map {:semitone [1]
                   :tone [0 1]
                   :minor-third [0 0 1]})
             flatten
             butlast
             (cons 1)
             cycle)))

(def major-scale
  (scale-from-tones
   [:tone :tone :semitone :tone :tone :tone :semitone]))
(def harmonic-minor-scale
  (scale-from-tones
   [:tone :semitone :tone :tone :semitone :minor-third :semitone]))
(def natural-minor-scale
  (scale-from-tones
   [:tone :semitone :tone :tone :semitone :tone :tone]))
(def locrian-mode
  (scale-from-tones
   [:semitone :tone :tone :semitone :tone :tone :tone]))
(def mixolydian-mode
  (scale-from-tones
   [:tone :tone :semitone :tone :tone :semitone :tone]))

(def scale-modes
  [[:major-scale          major-scale]
   [:harmonic-minor-scale harmonic-minor-scale]
   [:natural-minor-scale  natural-minor-scale]
   [:locrian-mode         locrian-mode]
   [:mixolydian-mode      mixolydian-mode]])

(db-rel semitone note-1 note-2)

(def keys-from-c
  [:C3 :C#3 :D3 :D#3 :E3 :F3 :F#3 :G3 :G#3 :A3 :A#3 :B3
   :C4 :C#4 :D4 :D#4 :E4 :F4 :F#4 :G4 :G#4 :A4 :A#4 :B4
   :C5])

(def semitone-facts
  (reduce
   (fn [db [note-1 note-2]]
     (db-fact db semitone note-1 note-2))
   empty-db
   (partition 2 1 keys-from-c)))
```

```clojure
(defne scaleo [base-note scale notes]
  ([note [1 . scale-rest] [note . ()]])
  ([note [1 . scale-rest] [note . notes-rest]]
     (fresh [next-note]
            (semitone note next-note)
            (scaleo next-note scale-rest notes-rest)))
  ([note [0 . scale-rest] notes]
     (fresh [next-note]
            (semitone note next-note)
            (scaleo next-note scale-rest notes))))

(defn key-restriction
  [instrument-state s1]
  (if-let [key (:key instrument-state)]
    (all (== key s1))
    succeed))

(defn scale-restriction
  [instrument-state scale-type]
  (if (:scale instrument-state)
    (all (membero [(:scale instrument-state) scale-type] scale-modes))
    succeed))

(defn cadence-restriction
  [instrument-state m7 s2 s4 s5]
  (case (:cadence instrument-state)
    :perfect   (all (== m7 s5))
    :plagal    (all (== m7 s4))
    :just-nice (all (== m7 s2))
    nil        succeed))

(defn- logic-program
  [instrument-state melody2]
  (fresh [melody
          m1 m2 m3 m4 m5 m6 m7 m8
          scale
          s1 s2 s3 s4 s5 s6 s7 s8
          base-note scale-type]
         (key-restriction instrument-state s1)
         (== melody [m1 m2 m3 m4 m5 m6 m7 m8])
         (== scale [s1 s2 s3 s4 s5 s6 s7 s8])
         (== m1 s1)
         (== m8 s8)
         (cadence-restriction instrument-state m7 s2 s4 s5)
         (== melody2 [m1 m2 m3 m4 m5 m6 m7 m1])
         (scale-restriction instrument-state scale-type)
         (scaleo base-note scale-type scale)
         (permuteo scale melody)))
```

```clojure
(defn compositions
  [instrument-state & [n]]
  (with-db
    semitone-facts
    (if n
      (run n [melody2]
           (logic-program instrument-state melody2))
      (run* [melody2]
            (logic-program instrument-state melody2)))))

(defn- random-composition
  [instrument-state]
  (rand-nth
   (or (seq (compositions instrument-state 1024))
       [[]])))

;; Loop

(defn- same-melody-params?
  [instrument-state-1 instrument-state-2]
  (let [non-melody-keys [:speed :gaps]]
    (= (apply dissoc instrument-state-1 non-melody-keys)
       (apply dissoc instrument-state-2 non-melody-keys))))

(defn composer-loop
  "Listens for new instrument states on instrument-state-ch and emits a
  random melody to melody-ch. The loop terminates when
  instrument-state-ch closes.

  Changes to :speed or :gaps does not compose a new melody, but alters
  the timing of the existing."
  [instrument-state-ch melody-ch]
  (go
   (loop [prev-instrument-state nil
          prev-composition      nil]
     (when-let [instrument-state (<! instrument-state-ch)]
       (let [gaps (for [i (range 8)] (get (:gaps instrument-state) i 0.5))
             speed (:speed instrument-state)
             new-melody (if (same-melody-params? prev-instrument-state
                                                 instrument-state)
                          (:melody prev-composition)
                          (random-composition instrument-state))
             new-composition {:gaps gaps
                              :speed speed
                              :melody new-melody}]
         (>! melody-ch new-composition)
         (recur instrument-state
                new-composition))))))
```

The system

The system



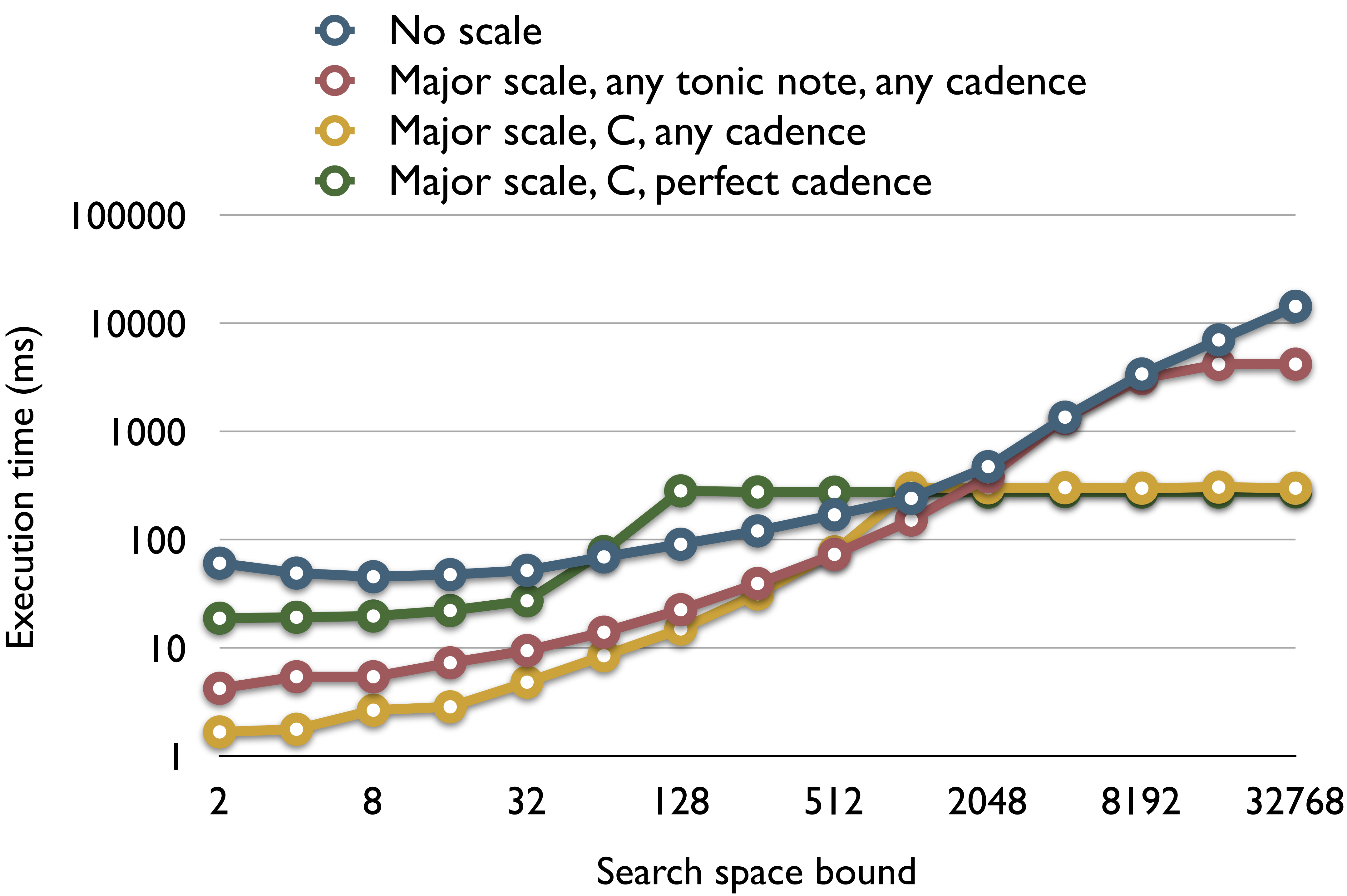
Experiments

- Goal: a reactive system
- Experiment 1: What is the size of the melody space and how long does it take to enumerate it?
- Experiment 2: What is a reasonable bound on the search space to achieve responsiveness?

Experiment 1

	No scale				Major scale			
	Any tonic note		C		Any tonic note		C	
	–	pc	–	pc	–	pc	–	pc
Melody space	25^8	25^8	25^8	25^8	$9,360$	$1,560$	720	120
Execution time (ms)	–	–	–	–	$4,299$	$3,852$	294	278
Melodies/second	–	–	–	–	$2,177$	404	$2,448$	431

Experiment 2



Conclusion

- Composer demonstrates it is possible to build a responsive interactive system with extremely small and succinct core
- The declarative nature of the core implementation makes it possible to extend the terminology to other types of music

Future work

- Proper sampling of search space
- Labeled interface
- Non-Western music
- User testing