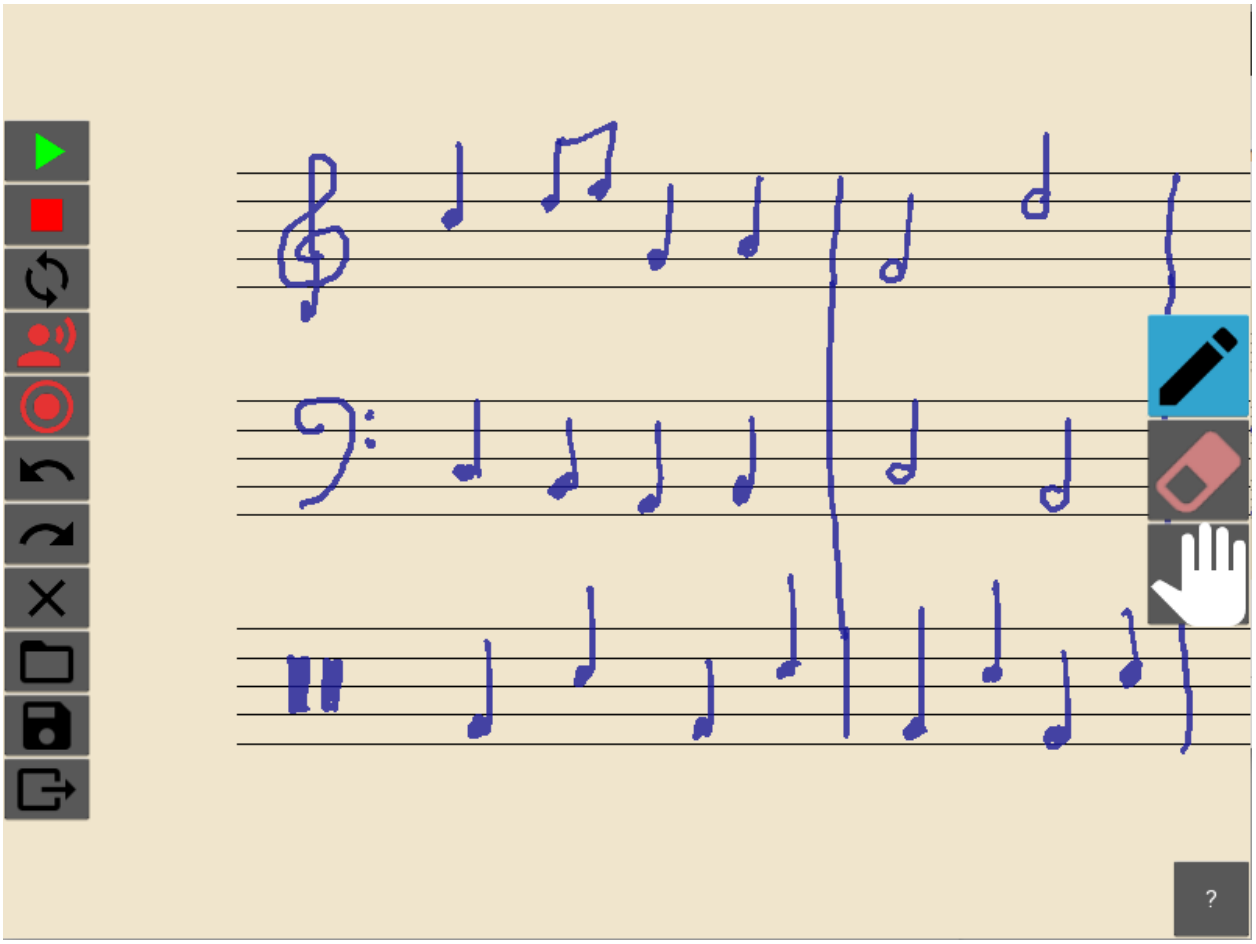


NotePad: Multimodal Music Composition

Ian Clester; Mergen Nachin

(Demo Video: https://youtu.be/4S_vnSqtDvA)

We built a music composition tool intending to enable people to enjoy the best aspects of composing on paper and composing with a computer. NotePad allows users to input musical ideas by writing (in a subset of traditional Western notation) or making noise (singing, humming, tapping). NotePad understands 21 musical symbols, including simple notes, rests, clefs, and a handful of instrument and chord designations, and supports transcribing melodies or rhythms one melody at a time.

The system operates on the principle that the internal state of the system should always match the user-visible state on the page. Thus, nothing is 'stuck' after being recognized, and the user can modify existing symbols, erase, undo, redo, and clear without the system ever becoming desynchronized from the page.

The system performs reasonably well for basic notating and rhythm transcription. Weak points include lack of expressiveness (there is no support for accidentals, key signatures, time signatures, or myriad other less common musical symbols, and we didn't have time to experiment with alternate sketch representations) and poor melody transcription performance.

Introduction and Overview

Written music dates back to at least 1400 BC, and for most of that history, several aspects have been less than ideal.[1] With pen and paper, hearing a composition requires the composer to stop and play it back on an instrument or rely entirely on the ear to audiate internally; neither option is accessible to novice musicians, and neither scales to large multi-instrumental works. And, as with all pre-computer writing, making edits and correcting mistakes is a tedious, laborious process.

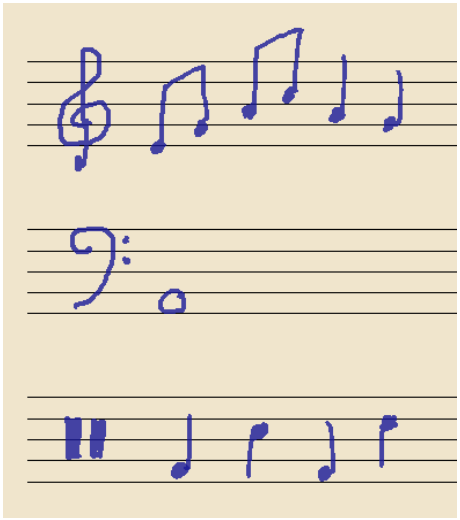
Today, there is plenty of powerful software for music composition and production, but interacting with the systems is often unnecessarily cumbersome. It lacks the intuitive ease of sketching down ideas on paper and requires the user to conform to the interface, rather than the interface conforming to the user. And going between pieces of software—e.g., from a DAW to a score editor—still involves unnecessary friction.

We hope to bridge this gap with NotePad, a tool to enable composing music using natural modalities (e.g., humming, singing, tapping, writing notation), while providing benefits afforded by modern composition/audio production software (playback, looping, undo, etc.).

System Description

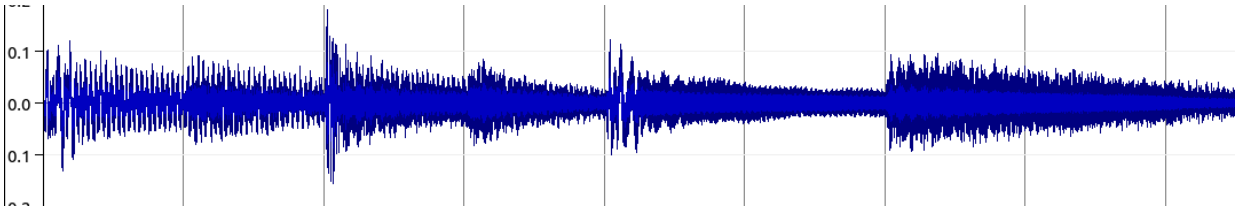
NotePad does three main things. It maintains a virtual canvas (the page of the notepad) for the user (and system) to draw on and segments and recognizes strokes on this page as musical symbols. It records and processes audio from the user, attempting to transcribe the melody or rhythm that it hears. And based on the page contents and recognition results, it builds up an internal musical structure in which it can playback and loop, even while the user continues to edit the content on the page.

As an example of the system's response, the following canvas state:



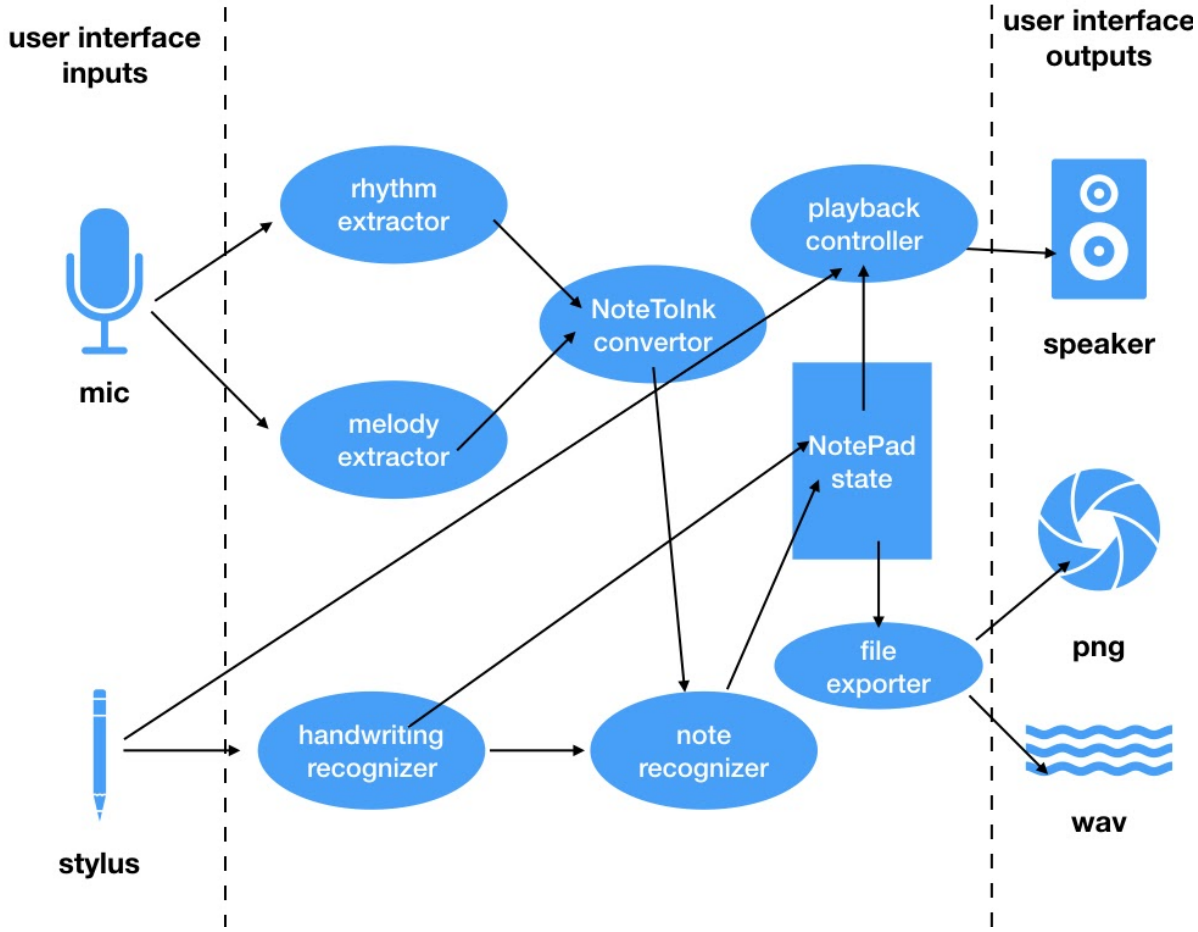
yields the audio response depicted in this waveform upon playback:

[1] https://en.wikipedia.org/wiki/Musical_notation#Ancient_Near_East



System Architecture & Evaluation

Here's a high level system architecture diagram.



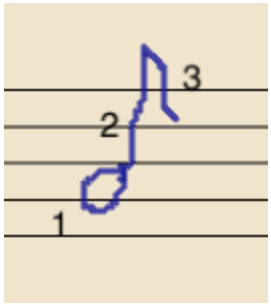
Handwriting Recognizer

As a reminder, user inputs hand-written symbols using a stylus on the surface (e.g., tablet) and the system needs to recognize the raw symbols/ink as musical notes and symbols and convert the input into an intermediate representation (e.g. 'eighthnote', 'trebleclef', 'halfrest'). Once it recognizes the raw input and understands this intermediate representation, it then either (1)

feeds the result to the ‘Note Recognizer’ part of the system for musical notes or (2) feeds the result directly into NotePadState for control symbols such as ‘trebleclef’ or ‘instrument-drum’.

Ink-based approach

We are treating the hand-written input as a set of strokes. We are not using the temporal information for the recognition. Please refer to the ‘Roadblocks and Pivots section’ for background on our decision.

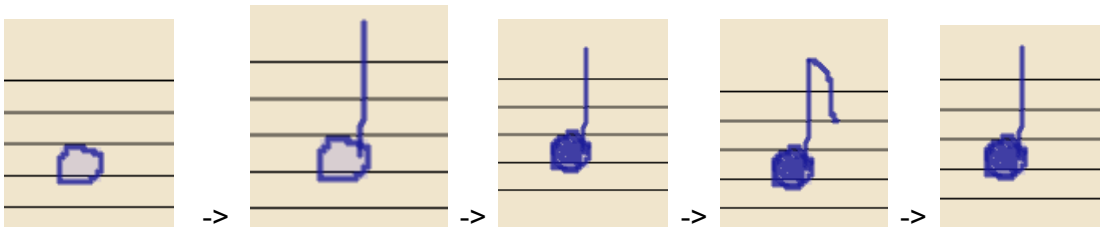


In the above example, the input is composed of three different strokes. A stroke is defined as a set of points. A stroke is considered finished whenever the stylus is released from the surface.

We then feed this data into an algorithm called [$P Point-Cloud Recognizer], discussed in more detail later. At a high level, $P is a template-based classifier. In other words, the user (or the developer of the system) trains by only providing a few examples of a class of symbols. In comparison to deep learning models where we provide a large number of training data, the template-based approach is much quicker to iterate and build on it. For instance, if the system miscategorizes a symbol, we can add it as a template to the database on the fly so that it categorizes it correctly in the future.

Segmentation

One of the main features is to incrementally edit a symbol. To illustrate this, here’s a sequence of inputs and corresponding recognition results.

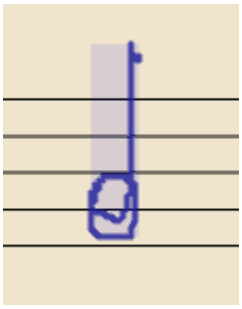


At each step, the user is adding a new stroke and the symbol is reclassified.
This feature is implemented is as follows:

1. Initially each ‘stroke group’ has a single stroke and associated bounding box with it.

2. Whenever a new stroke is added, it searches for other stroke groups where the bounding box is intersecting with the new stroke's bounding box. If so, add the new stroke to the 'stroke group'

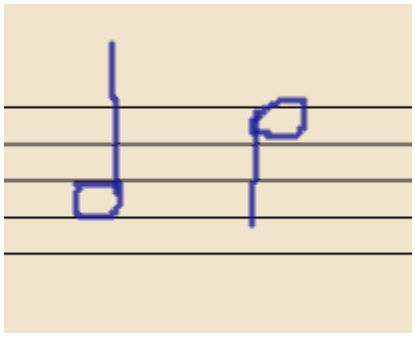
Below is an illustration of a bounding box (indicated as light blue).



Finally after the iteration above, we treat each 'stroke group' as a single entity and feed into the $P Point-Cloud Recognizer algorithm. This approach makes it possible to incrementally edit a symbol (e.g. add new strokes, delete arbitrary strokes etc).

Reversed stem scenario

We want to recognize a note that has either an upward stem or a downward stem.

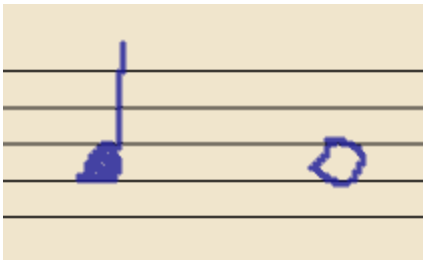


One naive approach would have multiple templates that are both upwards and downwards. However, that's not elegant because if we need to double the work every time we modify templates, and moreover, it won't be consistent, depending on whether a stem is upward or downward.

We solved by having a template only in one direction. During the recognition phase, we call the recognizer twice—once with the original input and second time with input 180 degrees rotated—and take the result with a higher score.

Note Recognizer

Once we know whether the note (represented as ink) is, say an 'eighthnote', then we want to figure out whether it is a middle C or a high G (or any other note). In order to achieve this goal, the system needs to know where the center of the note is. Consider the following example.



A naive approach would be taking the mean of all the points which would work in the whole note case (i.e., the right example); however, it won't produce correct results for notes with stems (i.e., the left example). The reason is that stems skew the mean, and in order to mitigate the issue, we incorporated an outlier elimination algorithm. We consider a point outlier if it is more than one standard deviation away from the mean. In pseudocode:

```
FindCorrectedCenter()
      For outlier_iteration: 1 to 10 (or until we have converged)
            Keep eliminating points that are considered outliers.

      Take the mean of the remaining points
```

With this approach, we were able to resolve the issue of skew. Once we know the “corrected mean“ of a note, we were able to locate which staff line the center is on, and consequently translate a gesture to, say middle ‘C’ or high ‘G.’

Transcription: Rhythm Recognizer, Melody Recognizer

For transcription, we relied on Essentia, an open-source library for audio analysis and music information retrieval. We chose this partly because of the apparent wealth of efficient algorithms and the promise of cross-platform execution. For both rhythm and melody transcription, we played four beats for the user and the recorded one measure of audio, compensating for audio system latency and applying Essentia’s algorithms to the recorded signal.

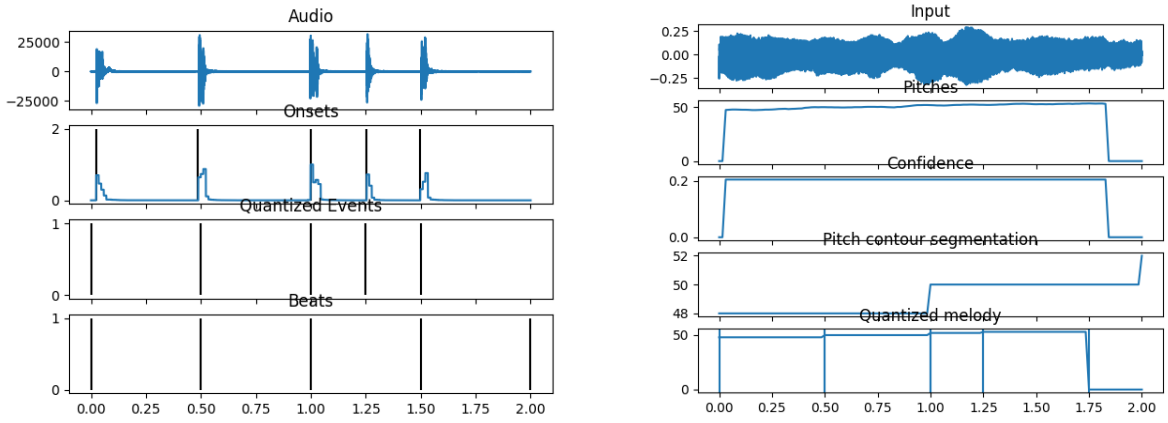
For rhythm recognition, we employed the High Frequency Content (HFC) onset detection function, which we found worked well in practice for sounds like clapping, tapping, and snapping. After determining the locations of the onsets in time, we quantize to the nearest eighth note based on the system tempo and return that set of events. The performance was generally satisfactory; four one measure of audio divided into eighth notes, there are only 256 (2^8) possible configurations of onsets, and the system seemed to distinguish between these without too much trouble.

Melody transcription unsurprisingly proved more challenging. We experimented with different Essentia algorithms and ultimately settled on PitchMelodia. This algorithm returns a pitch contour, which must be further segmented to produce quantized notes. Both parts of this process were fraught: especially for vocal recordings, pitch often wavers substantially. To address this, we quantized pitches to the key of C major; this was natural[2] because our system

[2] No pun intended.

does not support accidentals, but we could perform the same kind of quantization in the presence of a key signature. As with rhythm, we also quantized in time, sticking to units of eighth notes. But the Essentia algorithm frequently provided no estimate during pitched input, and despite our problem simplifications, the resulting transcriber was still pretty finicky.

The Essentia implementations provided a convenient starting point for us, but they are frustratingly opaque. We ultimately didn't have time to return to audio transcription once we had a passable proof-of-concept, but it's definitely something we would like to spend more time on to improve the system, perhaps implementing some of the relevant algorithms ourselves and tweaking as necessary to obtain better performance.



Debugging plots for rhythm (left) and melody (right) transcription.

Note-To-Ink Converter

This module takes a transcribed audio/rhythm as an input, and converts to an ink on the surface. Since it knows whether an individual element in the transcribed audio is, say 'quarternote', or 'halfrest', the system looks up from the template database directly and loads into data as strokes, and eventually rendering on the surface. Note that the template database is used as training data for the hand-writing recognizer, as well as, used as a database for providing inks/shapes for rendering.

Once we know the shape of the symbol to render, we need to figure the (x, y) position of the note. It uses `FindCorrectedCenter()` function again to align the y value on the staff. For x position, we are using the last non-filled position and keep filling the staff from left to right.

Using the same strokes from the template database gives us nice abilities to have the same infrastructure and tools to manipulate symbols. For instance, after transcription, if it got one 1 out of the 4 notes incorrectly, the user can edit that misclassified note directly either by erasing individual strokes or adding additional strokes.

After rendering on the surface is that, one additional step we are doing is calling NoteRecognizer again. This is a redundant and inefficient step because we are not learning anything new here (i.e., we already knew what note it is from the transcribed audio input). However, by doing this, we are making so that there's only a single entry point (i.e. NoteRecognizer) that changes state of NotePad state. Alternative way would have output of

transcribed audio (i.e. from Melody Recognizer) directly change the NotePad state. In a multithreaded environment, it's unsafe and brittle if we don't have proper synchronization points, and we are treating NoteRecognizer as such synchronization point. Moreover, it allows us to treat the NotePad state that is dependent only on the inks on the surface.

NotePad state

The surface contains only an ink. The NotePad state is dependent only on the inks on the surface.

NotePad state contains an internal note, and other internal states (e.g. staff instrument selection). Here's pseudocode that computes NotePad state from the surface.

```
interpret_surface(surface):
  strokes is a an array of singleton stroke
  while needs_merging():
     // update strokes
     do_necessary_merging_based_on_bounding_box()

  NotePadState = NoteRecognition(strokes)
```

Undo/Redo/Erase

For undoing and redoing, we can keep track and bookkeep stroke objects into undo and redo history. When we undo a change, we will delete the stroke back from the surface, and reconstruct the NotePad state using the same `interpret_surface()` function. Similar arguments apply for Redo and Erase.

Save/Load

For saving and loading, we can save only the inks on the surface. When we load back a saved result, we will load the inks back to the surface, but can reconstruct the NotePad state using the same `interpret_surface()` function.

Efficiency

Conceptually, we are re-evaluating every symbol on the surface and running the note-recognition module whenever there's a new stroke added to the surface. Benefits are state determinism that is only determined what's on the surface rendered and nothing else. There are concrete benefits as mentioned above in Undo/Redo/Erase and in Save/Load.

However, efficiency is an issue especially if there are many symbols on the surface. An acute observer would tell that whenever a new stroke is added, most results of note-recognizer won't be affected. In other words, only symbols which are close to the new stroke (e.g. bounding boxes intersecting) would be changed. Given this observation, we are able to use memoization

based on stroke ids. As a result of this, we are able to reduce computation necessary by 5.4x (one of our benchmark results was 1.9 seconds before but reduced to 0.35 seconds).

Playback Controller

Whenever the user clicks play, the playback controller looks at the internal NotePad state, which contains the transcribed notes and instrument selection. It then creates a new thread to play the audio by converting into midi.

On-the-fly-editing

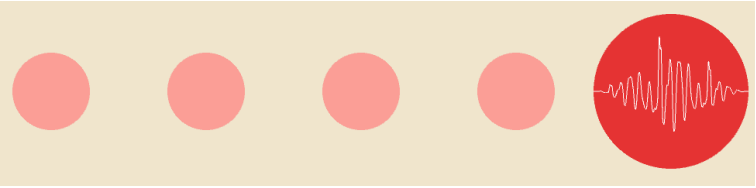
Since playback has its own thread **reading** from the NotePad state, and the surface has its own thread **writing** to the NotePad state, one nice feature we get is editing while play/loop is in progress and playback immediately incorporates the new changes without having to pause/stop. Thread safety isn't an issue because only one thread is writing/modifying the data.

Play-Pause-Loop-Resume

For play-pause-loop-resume functions, we are using the FluidSynth to schedule events. FluidSynth internal clock is used to schedule sequences of notes, turning them on and off at the correct time. When looping, the next sequence is scheduled in advance before the end of the current one, using a timer event that triggers a callback function. To pause, we record our position and remove future events from the sequencer.

Record Affordance and Feedback

One issue revealed in our user study was the lack of visual/audio feedback during the record melody/rhythm phase. They were confused about when the recording starts, and while the recording is happening if it's actually recording melody.



We solved this user experience issue by providing visual and audio feedback. Here is what it does:

- Use clicks Record Melody
- The system playback C note four times with the current tempo (e.g. 120 bpm)
- Each time it plays the note C, the red small dots become darker.
- At the fifth beat (also as C), the system starts recording as indicated by a big red dot.
- During the record mode, waveform representation of the input audio is shown inside the big red dot in real-time.

We believe that this approach solves a few things:

- Gives note calibration to the user before they hum.
- Give tempo calibration to the user.
- Give a sense of when to expect recording (i.e. after 4 beats, at the fifth)
- Give a sense of what is being recorded by showing the waveform.

Roadblocks and Pivots

There were some practical roadblocks we have encountered that are worth mentioning and the lessons learned along the way.

Mobile Deployment

After the user study with our prototype, we realized that the tablet peripheral we had been using for input simply would not cut it; new users had to look at the screen to see the state of the notebook, but write on a separate pad. Therefore, we prioritized getting our system to run on something with a touchscreen, and the most convenient device at hand was an Android smartphone. We made some of our early technology choices with portability in mind: Kivy advertises the fact that it can be deployed on mobile, and lower-level libraries (FluidSynth, Essentia) can be cross-compiled for mobile.

As demonstrated in our implementation demo, we successfully got our application to run on a phone, and we could use a cheap stylus to input notes and have the system recognize them. Soon after, we got audio output working by finding a FluidSynth binary precompiled for Android, and audio input and transcription were on the way.

And then, tragedy struck. Our development phone (also Ian's personal phone) bit the dust after 5 years of sometimes-acceptable performance. We tried deploying to another old Android phone (kindly lent to our cause) to no avail, due to version incompatibility issues with the FluidSynth library. As a result, we forged on with the desktop version of the project and were unable to demonstrate input via touchscreen in our final demo.

Lesson learned: Have a backup plan. If we had acquired a cheap Android tablet or attempted iOS deployment, we would have had another device to run on despite encountering a hardware failure.

Internet Outage

One day before the final video was due, Mergen's internet had an outage and we had to push-back some of our plans (e.g. video recording) and adjust our schedule (it was originally a tight schedule to begin with). We emailed the instructors right away. We had to wake up early and finish the video.

Lesson learned: It reminded us of "whatever can go wrong, will go wrong." It was a rewarding experience to pull it altogether despite the challenges, but we should keep in mind of striving to finish a day earlier to prepare for these unanticipated issues.

Pivot in Gesture-Recognition

Initially we performed stroke recognition Kivy's multi-stroke gesture recognition code[2], and we used a Kivy demo (available here) as the starting point for our codebase. This enabled us to get up and running quickly, and provided a very convenient interface for adding new gesture templates into the system.

However, we soon found that this interface was unacceptable for our use case. The main issue is that it takes a temporal approach, which (like the recognizer in Mini-Project 1) uses the order of the points into account and angular measurements in matching gestures with templates. In our application, there are several gestures that involve 'filling in' a patch of space, such as the notehead in a quarter note or eighth note, and the 'hat' or 'hole' in half and whole rests. For a recognizer that considers order and angle, gestures like these include a lot of noise, as the user scribbles in part of the gesture in a way that is neither consistent nor meaningful. After trying various fixes (tweaking recognizer parameters, adding more templates for the same gesture), we concluded that the basic approach was ill-suited to the gestures in our system.

Following this, we switched to a point-cloud representation which ignores point order and angle, found the $P recognizer (from the same group that created the $N recognizer, which Kivy's multistroke support is based on), and tried out a Python implementation called dollarpy.[3] The results were promising, especially with regard to the gestures that had proved so troublesome beforehand, and we soon completed the switch, adapting the existing Kivy infrastructure to work with this alternative gesture representation and recognizer.

User Study

We performed a user study during the Prototype stage and incorporated the feedback into our project.[3] However, since then, we haven't tested with a real user again. The following a summary of notes we took from this user study back in the Prototype stage, not the final stage. Along with the feedback, we are providing what we've done since to address the issues.

> Input device issues: user found drawing tablet peripheral confusing, and found mouse unwieldy.
>
> *Outcome: Made it possible to run an Android tablet with a stylus*

> We tweaked stuff up to the last minute and still left some minor bugs and debugging stuff in for the user test, which made it quite confusing.
>
> *Outcome: We made a UI rehaul to make it clean and minimal.*

[3]Prototype video, with user study: https://www.youtube.com/watch?v=9wlzd0ZgHYw

Some functionality was insufficiently discoverable (record rhythm & melody)

Outcome: We made the buttons more clear with a better icon.

User discovered Undo/Redo and used them without prompting

Comments that we anticipated: “How do I erase?”

Outcome: Created erase functionality

Comments that we didn’t:

“Why is it blue? Why is this one green?”;

Outcome: Made color consistent (BLACK for recognized, RED for unrecognized and BLUE for pending)

“Drawing rests is too hard; make it easier.”

Outcome: Showing a tutorial page in the beginning.

““Where did these notes come from?” during Record Melody/Rhythm phase

Outcome: Built a transcription affordance/feedback mechanism UI

Outcome: Built a feedback mechanism that shows audio waveform

Inconsistent system feedback

System displays recognition for some symbols (notes, rests) but not others (treble clef)

We threw on examples for the gestures, but should have included them for the symbols, too.

Unwanted system feedback: debug stuff, “red herring” data (stroke color)

Valuable feedback regarding gesture recognition, melody transcription.

We felt pretty good about the gesture recognition rate when we were testing it ourselves, but we’ve been trained by the system (and the drawing tablet peripheral).

Outcome: Showing a tutorial page in the beginning.

Recognize notes/rests too far from staff

Outcome: if a stroke is too far away from staff, then the system ignores it to recognize and leaves it

The user wanted to sign and export as a png

Outcome: Built export to PNG/WAV functionality

Limitations & Performance

Our system has many obvious limitations. Its support for Western notation is far from complete, with the lack of accidentals being probably the most egregious omission, but support for many missing elements would be straightforward to implement, given enough time. Here, we discuss

some of the more interesting limitations: things that would be harder for our system to support correctly, and why.

Compound Elements

Our system relies on symbols being spatially distinct: we segment strokes into gestures by testing for bounding box overlap. Many musical symbols, however, are *compound* in nature and visually connected. Prime examples are beams (grouping eighth notes and smaller subdivisions by connecting the stems), ties and slurs (grouping notes across time by connecting the noteheads), and chords (implicitly grouping notes vertically; may feature overlapping stems).

We implemented eighth-note beaming, but only for pairs; our approach relies on recognizing the pair of eighth notes as a distinct gesture, and then breaking it up into two notes. This approach is not scalable, as there is no limit to the length or variety of beaming constructions (especially when mixing subdivisions, or including rests). Similarly, there is no limit to the number of tied notes, and no (sufficiently small) limit to the size or variety of notes stacked in a chord. Offering complete supports for any of these would fundamentally require a more sophisticated approach to stroke segmentation.

Context-Sensitive Elements

Our system performs recognition with minimal context. Anywhere that symbols are recognized, any symbol may be recognized, and the system does not take into account any musical structure or existing symbols on the page during recognition. The system does take some context into account when *after* recognition, e.g. when assigning a note to the correct staff and pitch, when determining the active chord for a note, and when using clefs and instruments. But none of this influences the recognition itself, so the system may recognize symbols in nonsensical places: an instrument change written on the staff lines, or multiple barlines in succession with no notes or rests in between.

With context-sensitive recognition, it may be possible to improve recognition performance (meaningless possibilities are excluded, and context may decide ambiguous gestures) and more effectively support for more notation, such as ledger lines (which depend on how many other lines are above/below) and key and time signatures (which may only occur in certain places). As with compound elements, this would require an overhaul of segmentation, and it would additionally require significant changes to recognition.

Evaluation

We would have liked to compile test data and perform rigorous quantitative evaluations of the different components of our system. As time allowed, we mostly relied on qualitative observations from using the system (or watching our user interact with the system); these informed, for example, our switch for a temporal to spatial-only recognizer. In the absence of more informative figures, we might summarize the final performance of the system as ‘poor’ for melodic transcription, ‘pretty good’ for rhythmic transcription, and ‘decent’ for sketch recognition (within the supported notation).

Extensions

The immediate steps for improving the performance of our system are fairly clear: more sophisticated segmentation, context-sensitive recognition, and further experimentation and evaluation for melodic transcription. That said, the most exciting next steps may be in exploring and implementing new methods of musical sketch understanding, beyond Western notation.

Optical Music Recognition is a large, difficult problem, and the most effective way to deal with it—as relates to our original goal, in enabling users to express their musical ideas however they can—may be to partly sidestep it by exploring other kinds of notation. Piano rolls, pitch contours, waveform manipulation, and other kinds of graphical scores are musically interesting and perhaps more intuitive alternatives to Western notation that come with the benefit of a less complicated visual grammar. We're most excited about moving in this direction: implementing more kinds of sketch and audio understanding, and enabling the user to work with any and all of them, mixing and matching—supporting whatever way allows them to express their music best.

Tools & Libraries

See *notepad/requirements.txt* for a complete list of dependencies with exact versions we used. One can duplicate using `pip install -r requirements.txt`

Here are tools and libraries that are worth mentioning:

[1] Kivy - Open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps. (kivy.org)

[2] Kivy Multistroke gesture recognizer - We initially started with their example code as a prototype but towards the end, we had to strip away most features. The only things that were left were the gesture/template database and history manager. GestureRecognition was using a temporal based approach and we later realized that it's not very suitable we had to swap with DollarPy implementation. (https://kivy.org/doc/stable/api-kivy.multistroke.html)

[3] DollarPy - is a 2-D stroke-gesture recognizer designed for rapid prototyping of gesture-based user interfaces. By representing gestures as point-clouds, $P can handle both unistrokes and multistrokes equivalently.
(http://depts.washington.edu/acelab/proj/dollar/pdollar.html, https://pypi.org/project/dollarpy/)

[4] FluidSynth - is a real-time software synthesizer based on the SoundFont 2 specifications and has reached widespread distribution. Make sure to install version 2 or above.
(http://www.fluidsynth.org/)

[5] Essentia - Open-source library and tools for audio and music analysis, description and synthesis. We used this library for melody and rhythm extraction. (https://essentia.upf.edu/)

We also used **pyAudio** for recording mic input on desktop, and we were planning to use **audiostream** for the same thing on mobile.

Collaborations

We collaborated closely and often did pair programming throughout the semester. There are many parts of the code that boths of us touched. Here are non-exhaustive lists of features we have worked on individually.

Ian Clester:

- Audio playback, pause, loop, stop; FluidSynth integration.
- Audio recording and rhythm/melody extraction (pyAudio, Essentia).
- Main UI - application toolbars, write/erase/pan, icons
- Final recording feedback, waveform visualizer.
- Tutorial screen
- Mobile integration
- Debug mode
- Instrument selection, chords, beamed eighth notes

Mergen Nachin:

- UI Features such as play/loop/undo/redo/clear
- Save/Load/Export UI and functionality
- Converting midi format (melody) to ink (notes on the app)
- Gesture Recognition integration (dollarpy)
- Gesture segmentation
- Initial version of recording affordance
- Color consistency
- Various user improvements based on user studies

Videos for reference:
Design Studio: https://www.youtube.com/watch?v=WMKayevYdMY
Prototype: https://www.youtube.com/watch?v=9wlzd0ZgHYw
Implementation: https://www.youtube.com/watch?v=qWFNKth2o1w
Final: https://www.youtube.com/watch?v=4S_vnSqtDvA