Optional Assignment — Player Piano

Due: Monday, December 2nd, 5:00 p.m.

In this assignment, your mission is to implement a MIDI player that can play songs of your choosing and visualize the notes being played. MIDI files contain information about the notes that make up a song and are widely available on the internet for many different styles of music. (Try Googling “Super Smash Bros piano MIDI,” or even “Gangnam Style piano MIDI.”)

Towards the top of the program, a piano roll presents a rainfall visualization of the notes to be played. Piano rolls are a 19th century invention that used holes (called perforations) punched in a roll of paper to indicate when a note should be played. The roll of paper was slowly run through an array of sensors, and when a perforation reached a sensor, its corresponding note would be played. A music box uses a similar idea; a cylinder slowly rotates, and the positioning of pins (which collide with a metal comb) dictates the music that is played. In our representation, perforations are represented as rectangles on the screen that move downwards towards the keyboard. When a perforation reaches a key, the key is pressed for the length of the perforation. The longer a rectangle, the longer the note.

You can see the final product here: https://web.stanford.edu/class/cs106ax/piano-player/
It produces a visual similar to Synthesia, which you may have seen on YouTube (e.g. https://www.youtube.com/watch?v=gHKcUAMU2Lg).

In this assignment, we provide easy ways to read the MIDI data and play recordings of piano notes in your browser. Your job is to use an animation function to play notes at the appropriate times, and to display the graphical representation of perforations moving towards the piano. This program is complex, but this handout will walk you through a manageable approach to building it. By the end, you will have a MIDI player capable of playing any song – a feat you should be proud of!
Assignment Logistics

This assignment is new this quarter, and will probably replace HangKarel in the future. We are giving it out as an optional assignment this quarter to collect feedback. If you elect to complete this assignment, you have until Monday, December 2\textsuperscript{nd}. However, the assignment is not short, and it will not be easy to finish in a weekend, so we recommend you plan out a schedule. Set deadlines for yourself to finish each milestone.

If you complete this assignment, it will replace your lowest assignment grade. Additionally, to make it worth your time, we will increase your functionality grade on this assignment by two buckets (i.e. a ✓+ would be bumped to a ++).

Exploring the starter code

The starter code comes with a few files you should be aware of.

- The \texttt{piano-player.html} is, as usual, the entry point for this project. It references the files below, as well as a few extra libraries implementing functionality for reading MIDI files and generating piano sounds. We strongly recommend using Google Chrome for this assignment, since it has better compatibility with these libraries.
- The \texttt{js} directory contains the Javascript code for this project. \texttt{piano-player.js} is where you will implement all the code needed to finish this assignment. \texttt{util.js} contains some helper functions that are used by the starter code, but you do not need to understand or modify anything here.
- The \texttt{css} directory contains a CSS file that dictates the presentation of the web page (i.e. it codifies a dark background, centered content, etc). We will talk more about CSS in weeks 7 and 8. You do not need to understand or modify anything here.
- The \texttt{midi} directory contains a selection of MIDI files that you can use to test your project. You may find \texttt{test-simple1.mid} and \texttt{test-simple2.mid} in the \texttt{midi} folder useful for testing. The \texttt{test-simple1.mid} file plays the same note 4 times slowly, then plays one note higher 4 times slowly, then plays the original note 4 times again. The \texttt{test-simple2.mid} file plays ascending notes in the shape of a scale, then descends back down to the original note. The \texttt{test-velocity.mid} file plays a single note with increasing velocity, then decreasing velocity. (Velocity measures how hard the piano key is pressed.)
Milestone 1: Draw the piano

As a very first step, let’s draw the piano keys on the screen. A full-size piano has 88 keys, arranged like so:

There are 52 white keys corresponding to the 7 notes “A”, “B”, “C”, “D”, “E”, “F”, and “G”. The keyboard starts at “A” and repeats note letters every 7 keys, ending at “C.” Additionally, notes A, C, D, F, and G have black keys positioned just past them on the keyboard. Black keys are always positioned on top of white keys; take care to ensure that white keys are not stacked above black keys. You do not need to display the note letters anywhere in this assignment, but being aware of how the keyboard is laid out will help you understand the task at hand.

In this milestone, your goal is to implement the `drawPiano` function in the starter code to draw 88 `GRects` – one per piano key – and add them to the graphics window. This function should also return an array of the `GRects` that were added to the window, in left-to-right order (i.e. the leftmost “A” key should be first in the array, and the rightmost “C” key should be last). This will enable us to “press” a key by changing its fill color later on.

You should call `drawPiano` from the `PianoPlayer` function, passing it the `keysGw` graphics window. Note that, in this assignment, we are actually using two graphics windows. This allows for better performance in the final product. (For the curious: The graphics library redraws the entire graphics window whenever any one thing changes. By separating the two graphics windows, we don’t need to redraw the entire keyboard on every animation frame that shifts a piano roll perforation down the screen.)

Be careful to note that the keyboard ends on a white key; the last C on the keyboard should not have a black C# key following it.

When testing your code, you can load any .mid or .midi file from the “midi” folder to make the `GWindow` appear. The result should look as follows:
Milestone 2: Identify the notes playing at a point in time

For the next step of this assignment, you will need to implement a function (see `getNotesPlayingAtTime` in the starter code) that looks through the notes of a song and identifies the notes that are “on” at a given time.

The function is passed a song, and you can get an array of notes from `song.notes`. For each `note` object in that array, you can get the note’s start time (in seconds) as `note.start` and the note’s duration (also in seconds) as `note.duration`. Given the current time `time`, you should identify the notes that have started but have not yet stopped. Return an array containing these notes.

You can test your implementation by typing `testGetNotesPlayingAtTime()` into your browser’s Javascript console and pressing enter. This function is included in the starter code to test your implementation on a very basic song.

```
> testGetNotesPlayingAtTime()
All tests passed! You should be good to go on Milestone 2.
```

Milestone 3: Identify differences between two arrays of notes

Now that you can identify the notes playing at a position in time, you need to write some code to find the difference between the notes playing at two different points at time. This will help you in implementing milestone 4, when you will need to identify notes that should be playing now but were not playing previously (so that you can start playing those notes) and identify notes that should not be playing now (so that you can stop these notes).

For this milestone, implement the `getNewNotes` and `getStoppedNotes` functions in the starter code. These functions take two arrays of note objects: an array of notes that were playing previously, and an array of notes that should be playing now. The `getNewNotes` function returns notes that should be playing now but were not playing previously, and the `getStoppedNotes` function returns notes that were playing previously but should not be playing now.

Try to structure your solution so that you avoid duplicating large amounts of code. You only need to write a few lines of code for this milestone.

Milestone 4: Play piano sounds

Now it’s time to play some music! Chrome (and other browsers) offer a toolkit called the “Web Audio API,” and we have bundled two libraries (Tone.js and @tonejs/piano) into the starter code that make this functionality easy to use.

Each note object (in the `song.notes` array, and in the array that is returned by your Milestone 2 `getNotesPlayingAtTime` function) contains a `noteName` property specifying which pitch should be played, as well as a `velocity` property specifying how hard the piano key should be pressed.

You can play a note like so:

```
piano.keyDown(note.noteName, piano.immediate(), note.velocity);
```

To stop the note, you can use this code:

```
piano.keyUp(note.noteName);
```
In this milestone, you should write an animation function that keeps track of the time elapsed since the beginning of the song, and uses `getNotesPlayingAtTime()` to identify which notes should be playing. You will need to keep track of which notes were playing the last time your animation function was called, and use your code from Milestone 3 to `piano.keyUp()` and `piano.keyDown()` appropriately.

In order to start and stop notes at the appropriate time, you will need to keep track of how much time has passed since the song started playing. To make this easy, we have included a convenient `Stopwatch` class. You can create a `Stopwatch` initialized at time $t = 0$:

```javascript
let stopwatch = Stopwatch();
```

Then, use `stopwatch.start()` to start counting time, `stopwatch.stop()` to pause the stopwatch, and `stopwatch.getTime()` to see how many seconds have elapsed on the stopwatch. In theory, you should be able to count the number of times your animation function has been called and use that number to determine the number of seconds that have passed. But the timer event heartbeat isn’t as precise as it could be, so it’s better to use the dedicated `Stopwatch` class for timing information.

You should start playing music when the user clicks the `keysGw` graphics window. (The piano keyboard is stacked on top of the `GWindow` for the piano roll, so be sure to add the click event listener to this `GWindow`.) When the user clicks, you should start a `Stopwatch` to track how much time has passed in the song, and you should start the animation function to begin playing notes. If the user clicks again, you should `pause` the song. Clicking again should resume the song, and clicking once more should pause once more.

You may find `test-simple1.mid` and `test-simple2.mid` in the `midi` folder useful for testing. The `test-simple1.mid` file plays the same note 4 times slowly, then plays one note higher 4 times slowly, then plays the original note 4 times again. The `test-simple2.mid` file plays ascending notes in the shape of a scale, then descends back down to the original note.

**Milestone 5: “Press” keys on the piano**

Now, you can update the graphical display to show the piano keys being pressed while the notes are played. In the previous milestone, you used `note.noteName`, passing it to the `piano` to start or stop a pitch. Each note also has a `noteIndex` property, which is the index of the key on the piano that plays this note. Assuming Milestone 1 is implemented correctly, when playing a note, you can index into the array returned from `drawPiano` to get the `GRect` representing the corresponding piano key. Every time you call `piano.keyDown` to start playing the sound for a note, you should also call `setFillColor` on the corresponding `GRect` to change the key’s color to `PRESSED_KEY_FILL_COLOR`.

Un-pressing the key is more difficult. The array returned from `drawPiano` contains both white and black keys; when un-pressing a key, you need to know whether to restore its fill color to white or black.

To address this, you should modify `drawPiano` so that it adds a `fill` property to each `GRect`, containing the original fill color:

```javascript
keyRect.fill = "White";  // or "Black", depending on the key
```
Since this fill property is something you have tacked on and isn’t part of the graphics library, its value will not change if you call setFillColor to “press” the key. Therefore, when you want to un-press the key, the fill property will still contain the key’s original color. You can pass this value to setFillColor to un-press the key.

By the end of this milestone, your piano should play notes out loud when you click it, and you should see those notes highlighted on the keyboard while they are playing.

**Milestone 6: Identify the perforations visible at a point in time**

Once your piano is playing notes, it’s time to draw the perforations on the screen that show which notes are about to be played. To simplify this task, we would like you to first implement a getNotesVisibleAtTime function that is similar to the getNotesPlayingAtTime function you implemented in Milestone 2. Whereas the latter function returns notes that are playing at a given time, this function should return an array of notes whose perforations are visible at a given time.

The key intuition here is to realize that times can be mapped to y-coordinates. The current time (provided to the function as the time argument) maps to the top of the keyboard, since notes at that time have “hit” the keyboard and are currently playing. Notes later in time are further up the screen. The PIXELS_PER_SECOND constant (set to be 128 in the source code) specifies how far up the screen later notes are; a note starting 1 second after the current time should appear 128 pixels above the top of the keyboard.

In implementing this function, you should determine which notes have perforations that would be visible on the screen at a given time. Add these notes to an array and return that array. Be aware that a note may only be partially visible (e.g. it may be the case that only the start of the perforation is visible on screen); these notes should still be added to the array.

As with Milestone 2, we have provided a testGetNotesVisibleAtTime() function, which you can call from your browser’s Javascript console.

**Milestone 7: Draw the perforations for time \( t = 0 \)**

In this milestone, you should modify your PianoPlayer function to show the starting note perforations when a MIDI file is loaded. (These perforations should show before the user clicks the piano.) Call the getNotesVisibleAtTime() function you just implemented to get the notes visible at time 0. For each of these notes, you should create a GRect representing the perforation. The GRect should be created as follows:

- The GRect should have a width of PERFORATION_WIDTH and a height based on the note duration and PIXELS_PER_SECOND.
- It should be horizontally centered above the piano key that it is meant to play (remember, you can get the key GRect from the array of keys from Milestone 1, and GRects have getX() and getY() methods). It should be vertically positioned so that the bottom of the perforation is spaced from the top of the keyboard based on the number of seconds until the note will be played.
- The fill color should be set based on the note’s velocity (i.e. how hard the piano key will be pressed). We have provided a getColorForVelocity() function that returns a color when passed note.velocity.
You can test this milestone using `test-simple1.midi`, `test-simple2.midi`, and `test-velocity.midi`. Your graphics should appear like this:

Milestone 8: Animate the perforations moving towards the keyboard

In Milestone 3, you wrote code to find the difference between the notes that were playing in the last animation frame and the notes that should be playing at the current time. For new notes, you used `piano.keyDown()` to trigger new sounds, and for notes that should no longer be playing, you used `piano.keyUp()` to stop the sound.

In this milestone, you will do something similar. From your animation function, you should call `getNotesVisibleAtTime()` to determine the notes that should be visible at the current time. For newly-appeared notes, you should create new `GRects` the way you created them in Milestone 6. (Be sure to structure your code so that it is not duplicated between the `PianoPlayer` function and your animation function.)

For notes that should no longer be visible, you should remove the corresponding `GRect` from the `GWindow`. In order to do this, you may want to maintain an array of `GRects` that are currently on the screen; that way, when you identify a note object that should no longer be displayed on the screen, you can find the corresponding `GRect` in the array and remove it from the `GWindow`.

Finally, for notes that were previously visible and still are visible, you should update the locations of the corresponding `GRects` so that they animate a few pixels down the screen, towards the piano keyboard. You should recalculate the position of the `GRect` for each of these notes based on the current time, and use `rect.setLocation(x, y)` to update the perforations’ locations.

When implementing this milestone, you should reuse helper functions that you wrote in Milestone 3 to determine which notes are new and which notes have gone off screen.

Milestone 9: Add pedal

On a piano, a note is played for as long as the key is pressed. However, if the sustain pedal is pressed while a note is being played and then the key is released, the note continues to play until the pedal is released. To make this extra-clear for those who have not played a piano, imagine a person presses the C key, presses down on the sustain pedal, releases the C key, and then releases
the sustain pedal two seconds later. The note C will be played from the time the C key is pressed until the sustain pedal is released.

In this assignment, you will use information in the `song.pedalChanges` array to trigger the pressing/release of the sustain pedal. The array contains objects; each object has a `start` property indicating when the pedal is pressed or released, as well as a boolean `down` property indicating whether the pedal is pressed down or released at that time. For example, consider this example `pedalChanges` array:

```
[{
    start: 1,
    down: true,
}, {
    start: 2,
    down: false,
}, {
    start: 3,
    down: true,
}]
```

The pedal always begins in the “up” position. 1 second into playback, the pedal is pressed down; 2 seconds into playback, the pedal is released; and from 3 seconds until the rest of the song, the pedal is down.

In this milestone, you should write code similar to what you did in milestones 2 to identify whether the pedal should be down at a certain point. Then, add code similar to milestone 4 that calls `piano.pedalDown()` to press the pedal down and `piano.pedalUp()` to release the pedal.

The `test-pedal.mid` file may be useful for testing. This file plays a note 4 times without pedal, then 4 times with pedal, then 4 times without again, followed by 4 times with again.

When this milestone is finished, you should notice a significant difference in how certain songs sound (especially slower songs). `Clair de Lune.mid` sounds choppy and mechanical without pedaling, but should sound beautiful once pedaling is added.

**Congratulations!** You are finished! You should be left with a final product that you can be proud of. Try downloading some other MIDI files to see what else your player piano can play. If you are up to the task, see if you can think of ways to add new, creative features. For example, you might make the piano keys clickable, so that users can play sound by clicking on the piano.