A LEVEL COMPUTING PROJECT:

PROJECT // NOTEBLOCK

PROJECT DOCUMENTATION

MOSES SUNG HIM NG
CANDIDATE NO.: 5694

PROJECT ANALYSIS

Introduction

The tool of choice for most music producers is the digital audio workstation (DAW), a piece of software in which you can record, edit and arrange music. Technologies such as MIDI and Virtual Studio Technology (VST) plugins even make it possible to produce an entire track from start to finish on your computer without using any recorded audio.

Another way of making music on a computer is using the sandbox block game Minecraft, in which it is possible to create circuitry with redstone and use it to activate note blocks which play musical notes. Surprisingly complex arrangements can be achieved in this manner, due to the ability to select different instrument sounds and the way that the position of a note block relative to the player determines the volume and pan of the resulting sound. However, laying down the note blocks and circuits in-game is tedious and time-consuming.

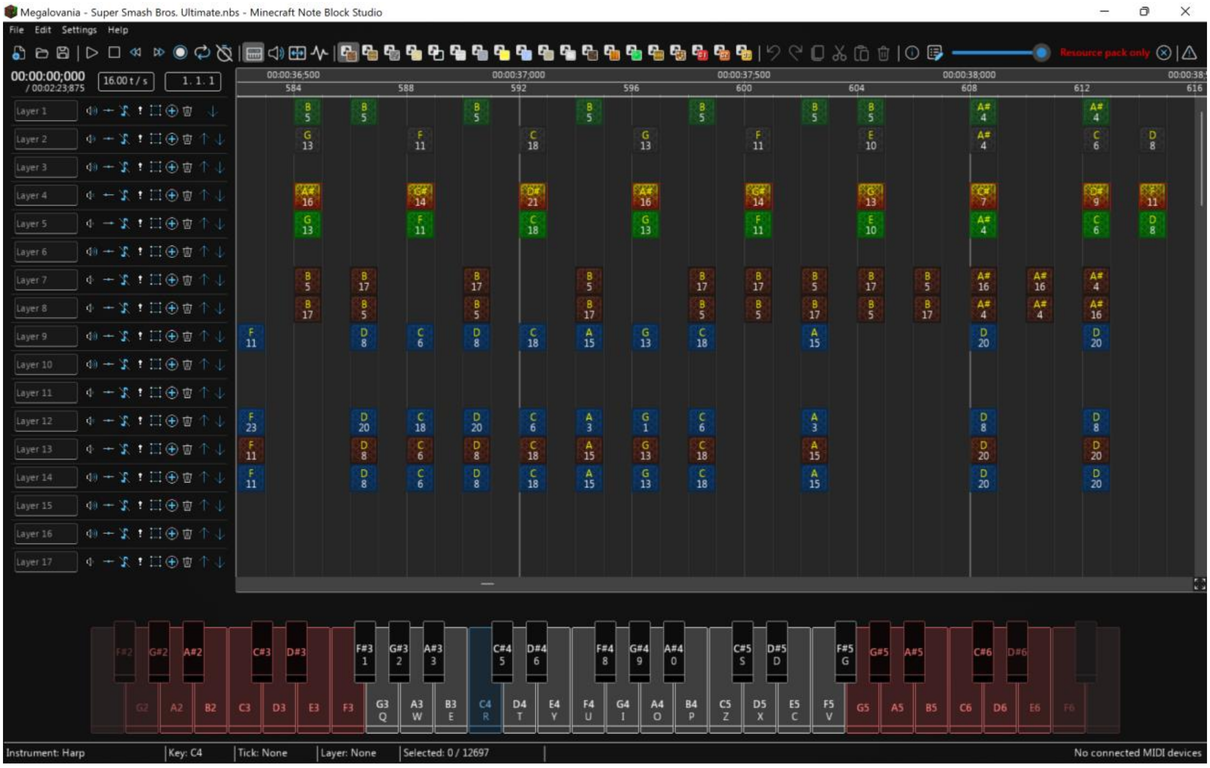
The aim of my project is to unite these two worlds. My program is essentially a DAW, with the catch that you can only work with the sounds of Minecraft note blocks. It includes features commonly found in DAWs such as a MIDI-like pattern editor, sequencer, effects and live playback.

On a surface level, this simply makes it easier to produce music that has Minecraft's unique aesthetic. But more important, and what makes this project different from other DAWs, is the ability, as a consequence of the limited sound palette, to export songs into a format (known as a "schematic") that can be placed and played back within the game itself. In fact, the primary goal of my project is to offer a streamlined, optimized workflow for creating music to be played within Minecraft, as an alternative to the inefficient and inflexible process of manually placing blocks.

Research

Open Note Block Studio

https://opennbs.org/

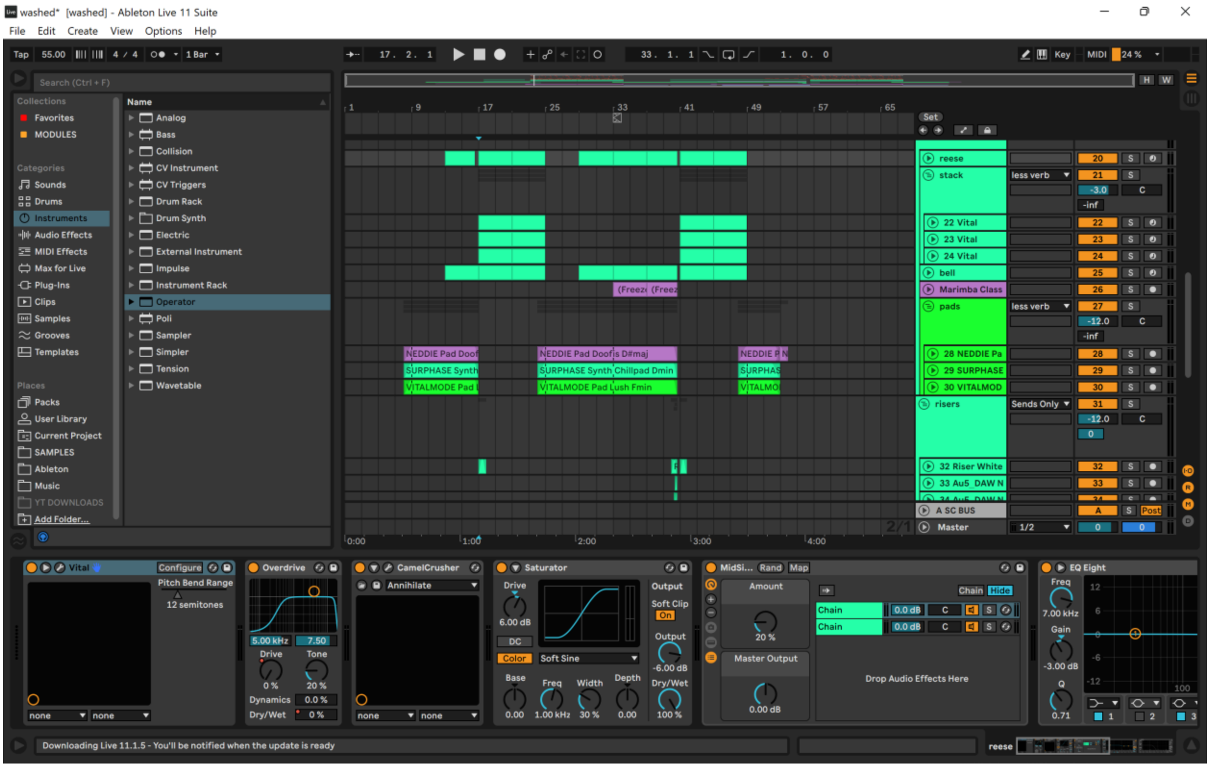


The song shown in this screenshot is a cover arranged by ShinkoNet.

Open Note Block Studio is an open source note block song editor popular among Minecraft musicians (such as ShinkoNet, who makes music professionally for Hypixel, the largest Minecraft server worldwide). It has features such as channels, volume/pan, and even features that my project does not have, such as custom instruments, non-vanilla tick speeds (comparable to tempo) and the ability to export songs as datapacks. However, the interface is awkward, features such as MIDI-like pattern editing and real-time effects are absent, and volume/pan settings do not carry over into exported in-game layouts.

Ableton Live 11 Suite

https://www.ableton.com/en/

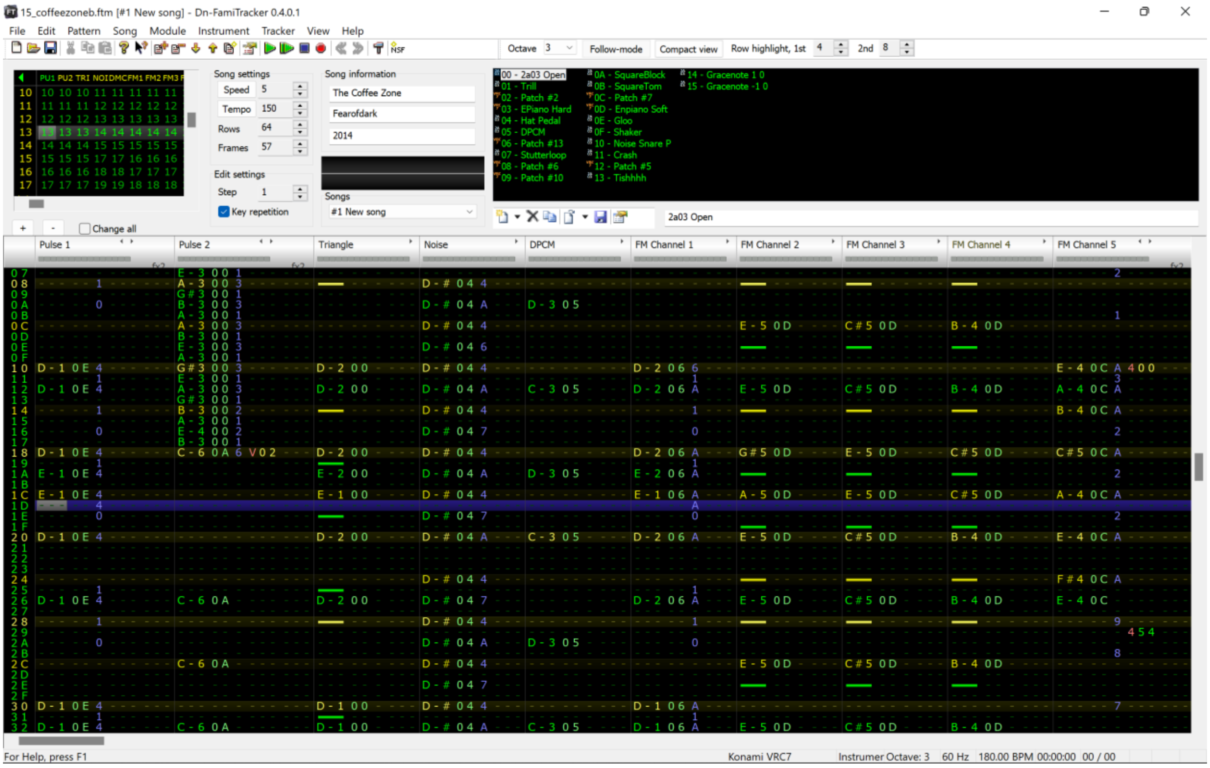


The song shown in this screenshot is a song I wrote myself – youtu.be/JZk6PqHJ1iw

Ableton Live is a mainstream DAW which I personally use for music production. It was originally designed solely for live performance, but has since expanded into a fully-fledged music production software suite. It is much more powerful than my project, with capabilities such as loading VST plugins, working with audio or MIDI, making track groups and return tracks, and even the ability to code your own plugins entirely within the DAW using the visual programming language Max/MSP. Much of my project's interface is modelled after this software, such as the effect rack at the bottom of the window, which is a very convenient feature that most other popular DAWs do not have.

Dn-FamiTracker

https://github.com/Dn-Programming-Core-Management/Dn-FamiTracker



The song shown in this screenshot was composed by Fearofdark.

Dn-FamiTracker belongs to a family of software called music trackers, which are the predecessors of DAWs and operate in a text format, as can be seen in the screenshot above. This particular tracker is designed for creating music to be played on the Nintendo Entertainment System (NES), a video game console dating back to the 1980s. The pattern system in this software, where changing the contents of a pattern affects all of its instances in the sequencer, is a system similar to what I will be using in my project, and has also been adopted by more powerful DAWs such as FL Studio.

Sponsor

My client is Isaac Taylor-Cummings, a fellow student. His hobbies include playing the piano, producing music in GarageBand, and playing Minecraft. He requested a DAW-like program for making music for Minecraft, but left me to figure out the details, of which I then asked his opinion in the following interview.

Interview

Me: So, what genres of music do you like? Whether that be to listen to or to make.

Isaac: I mostly enjoy listening to classical hip hop and R&B, from the late 90s and early 00s. But I also listen to a lot of lo-fi music, and in terms of music I make, it's usually a mix of those two styles.

Me: When making music, what parts of the process do you find yourself spending the most time on?

Isaac: I would say that I spend the most time on sound design – picking out instruments and samples, tweaking and layering them in interesting ways. But I will often also record myself, playing the piano for example.

Me: So, what would you expect from a program for making music for Minecraft?

Isaac: I think user friendliness is very important; the software should be pretty easy to use – certainly easier than just working in-game, so there is a reason to use it. For example, you should be able to easily repeat melodies or sections of a song; to do that in Minecraft you either have to mess with complicated redstone circuitry or resort to laying down all the note blocks a second time.

Me: Do you think it would be helpful to have a piano roll to input notes, similar to GarageBand?

Isaac: For sure it would! Inputting notes in game is annoying and slow – if you mis-click a note block once you have to cycle through all the pitches again to get back to the right one.

Me: A feature I have planned is the ability to export a song as a Minecraft schematic. When it comes to that, would you prefer a compact in-game layout, or control over the volume and position of sounds?

Isaac: A compact layout could be nice, as it would be more practical to build in survival mode in terms of resources. However, compact redstone builds can become complicated very quickly and I think that they are possibly better suited to manual design and optimization. In that light, I would say precise control over sounds is a pretty good alternative and would eliminate much of the tedium of doing the same thing in game.

Me: How about other export formats? Do you think it would be useful to be able to export as an audio file, for example?

Isaac: Yes, I think so; it would let me listen back to songs I've made without having to open Minecraft or your software every time. Being able to export each instrument or voice in a song as an individual audio file would also be nice, as I could use them to continue work on a song in other audio software if I decide that I want to take it beyond Minecraft.

Me: Are there any other features that you would want in particular?

Isaac: I've seen some note block songs on YouTube, where they use lots of note blocks to create effects such as sustained notes and echoes, and I think it would be very useful to have a way to do those things automatically.

Interview analysis

Throughout the interview, my client emphasized the importance of a user-friendly interface and the acceleration of processes which would be tedious and repetitive to do in Minecraft itself. A system similar to that of FamiTracker or FL Studio, where "patterns" of notes are defined independently from the structure of the song to then be placed within it, would make it easy to repeat any part of a song, whether that be a single instrument or an entire section. To achieve effects such as sustained notes and delay effects, I will be making use of an effect rack system similar to Ableton's, where the settings of every effect on a channel are accessible at once. My client also requested the ability to export the audio of individual voices or instruments – in my opinion this is best accomplished on a per-channel basis, also known in production circles as exporting "stems".

Music in Minecraft

A key aspect of the project is that only sounds from Minecraft can be used in a song, and only in a way that respects the constraints of the game. Hence, a brief discussion of how music creation works within Minecraft itself is warranted, to justify some of the design decisions that I will be detailing later in the documentation.

Minecraft is a procedurally generated sandbox game, in which the game world is organized into a three-dimensional grid of "**blocks**". Musical sounds in Minecraft are produced by a special type of block, called a "**note block**". A player can trigger a note block to play its specified sound by hitting it in-game, or by feeding a "**redstone**" signal into it. Redstone is essentially the electricity of Minecraft – signals can travel down wires made of redstone to trigger a wide variety of blocks, including note blocks. Various redstone components also exist to manipulate redstone signals; for example, a "repeater" delays a redstone signal by a specified number of redstone ticks, and an "observer" sends a redstone signal whenever any change is made to the block in front of it. Among other things, redstone can be used to construct logic gates, and some players have managed to build entire working computers within the game.

The game state updates 20 times a second, and each of these updates is called a "**game tick**". Redstone circuitry usually updates only 10 times a second; in other words, a "**redstone tick**" is the same length as two game ticks. However, there exist tricks that allow you to generate two parallel redstone signals one game tick apart from each other, which ultimately makes it possible to trigger note blocks 20 times a second.

Note blocks can be set to a number of different instruments which each have their own unique sound, such as drums, flute and guitar. Each note block also can be assigned a pitch within a two-octave range; this range always starts on an F# note, but different instruments have higher or lower ranges.

Objectives

Italics indicates an aspirational objective

1. **General interface**
    a. UI layout:
        i. Top menu bar for file I/O and settings
        ii. Play, pause and navigation buttons
        iii. BPM and time signature settings
        iv. Pattern bar on left
        v. Instrument and effect bar at bottom
    b. Clean, modern interface
        i. Modern UI theme
        ii. Hi-DPI display support
        iii. UI adapts effectively to resizing of window
    c. Resizable UI sections
    d. Alternate mixer view
2. **Instruments**
    a. Access to all default Minecraft note block sounds
    b. Select a single instrument per channel
    c. Select a main and sustain instrument per channel
    d. Custom note block sounds
    e. FamiTracker-style instrument customization:
        i. Attack, sustain and release sections
        ii. Per-instrument volume automation
        iii. Per-instrument pan automation
        iv. Ability to save instrument presets
3. **MIDI-like pattern editor**
    a. Simple scrollable and editable piano roll
    b. Per-note sustain setting
    c. Play notes as they are selected/edited (can be disabled)
    d. Arbitrary pattern lengths (requires arbitrary pattern placement)
    e. Support for polyphony
4. **Sequencer**
    a. Simple scrollable grid-constrained sequencer
    b. Channel headers
        i. Per-channel volume and pan settings
        ii. Per-channel mute and solo buttons
        iii. Click to open per-channel instrument/effect settings
    c. Playback loop markers
    d. Drag and drop pattern placement
    e. Arbitrary pattern placement
    f. Pattern preview inside sequencer
5. **Effects**
    a. 3 effect slots per channel
    b. Drop-down effect menus to add effects to slots
    c. Ability to swap effects between slots/delete effects from slots
    d. UI sliders and knobs for effect control
    e. Visualizers for selected effects

    f. *Unlimited, scrollable effect slots per channel*

6. ***Automation***
    a. *Discrete, per-tick automation*
    b. *Linearly interpolated automation*
    c. *Per-channel volume and pan automation*
    d. *Per-channel effect automation*
7. **Playback**
    a. Smooth, low-latency song playback
    b. Per-pattern playback
    c. *Ability for live edits to affect playback*
8. **File I/O**
    a. Custom file format for saving and loading songs
    b. Ability to export to WAV audio files
    c. Per-channel audio export (creates stems to be used in other DAWs)
    d. *Ability to export to other audio file formats*
    e. *Ability to import patterns from MIDI*
9. **In-game implementation**
    a. Ability to export to Minecraft schematic files
    b. Lag-efficient in-game layout
    c. 20 ticks/second playback (possible with some tricks)
    d. Automatic removal of unused redstone lines
    e. In-game layout load calculation:
        i. Channel-based calculation
        ii. Pattern-based calculation
        iii. *Per-tick calculation*
    f. Spreading of high-volume channels onto multiple quieter lines
    g. *Smart layout calculation for volume/pan automation:*
        i. *Diagonal redstone lines*
        ii. *Binary redstone line stacking*

Modelling

The following are ideas for some classes which I think I will need – they are based around concepts seen in DAWs such as Ableton and FL Studio.

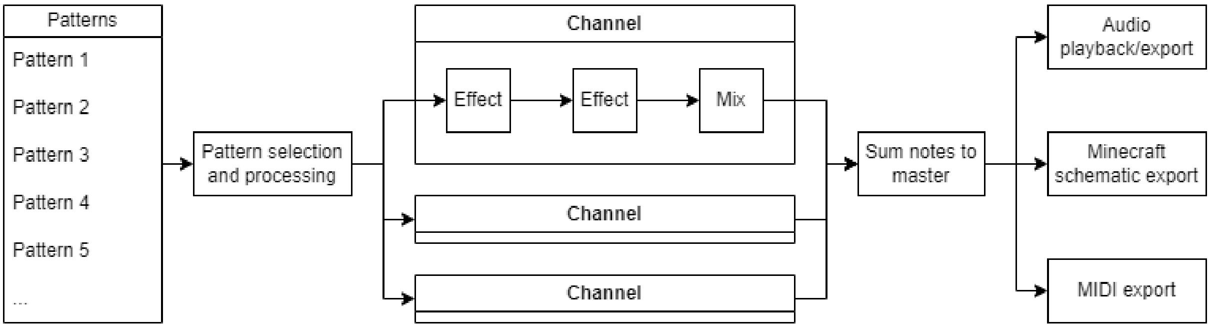
Note	A class representing a note from a Minecraft note block.
+ instrument: Instrument	The instrument of the note – a reference to a shared Instrument instance.
+ pitch: int	The pitch of the note – from 0 to 24 corresponding to the possible note pitches in-game.
+ volume: float	The volume of the note – from 0 to 1.
+ pan: float	The stereo pan of the note – between -1 (left) and 1 (right).
+ sustain: int	Metadata created by **Pattern**, which can be used by **Effect** to convert a note into a sustained stream of notes.
+ is_main: bool	Metadata created by **Effect** which can be used by other instances of **Effect** to selectively apply effects to notes.
+ altered_copy(**kwargs) -> Note	Returns a copy of the note with altered attributes as specified by keyword arguments.
+ stereo_volumes() -> np.ndarray	Returns a numpy array which can be multiplied with an array of audio to achieve the desired volume and pan.

<<interface>> **Pattern**	An interface for patterns – sequences of notes which can be placed into channels in the song arrangement.
+ get_notes(timestamp: int) -> list[Note]	Takes the current timestamp adjusted relative to the position of the pattern in the arrangement as an argument. Returns a list of the note(s) in the pattern at that timestamp.

<<interface>> **Effect**	An interface for effects, which can be added to a channel and used to sequentially modify a list of notes. This makes things such as note sustain, delays and volume/pan effects possible.
+ tick(notes: list[Note]) -> list[Note]	Accepts a list of notes and returns a new, modified list. Effects are timestamp agnostic, which allows for features such as arrangement-wide loops.

Channel	A class which gathers patterns and effects together into one entity which can be used by the playback engine.
+ patterns: list[tuple[Pattern, int]]	A list of the channel's patterns as well as the times at which they appear in the arrangement. This is necessary as one pattern can be used at multiple times or even in multiple channels.
+ effects: list[Effect]	A list of the channel's effects. Notes from the pattern are run through every effect in the list sequentially every tick.
+ get_notes(timestamp: int) -> list[Note]	Accepts the current timestamp from the playback engine and returns the channel's notes output at that timestamp.

Below is a diagram showing the possible signal flow between these classes and other components. The **Note** class is the primitive unit of data which is passed around and processed.



PROJECT DESIGN

Song structure

At the heart of my project is a data structure representing a **song**, which can be edited and played back within the program. I designed this based on the structures and workflows of the digital audio workstations that I am familiar with, most notably Ableton Live and FL Studio, while also respecting the technical constraints imposed by Minecraft. Much of my project is built around reading from and writing to this data in complex ways.

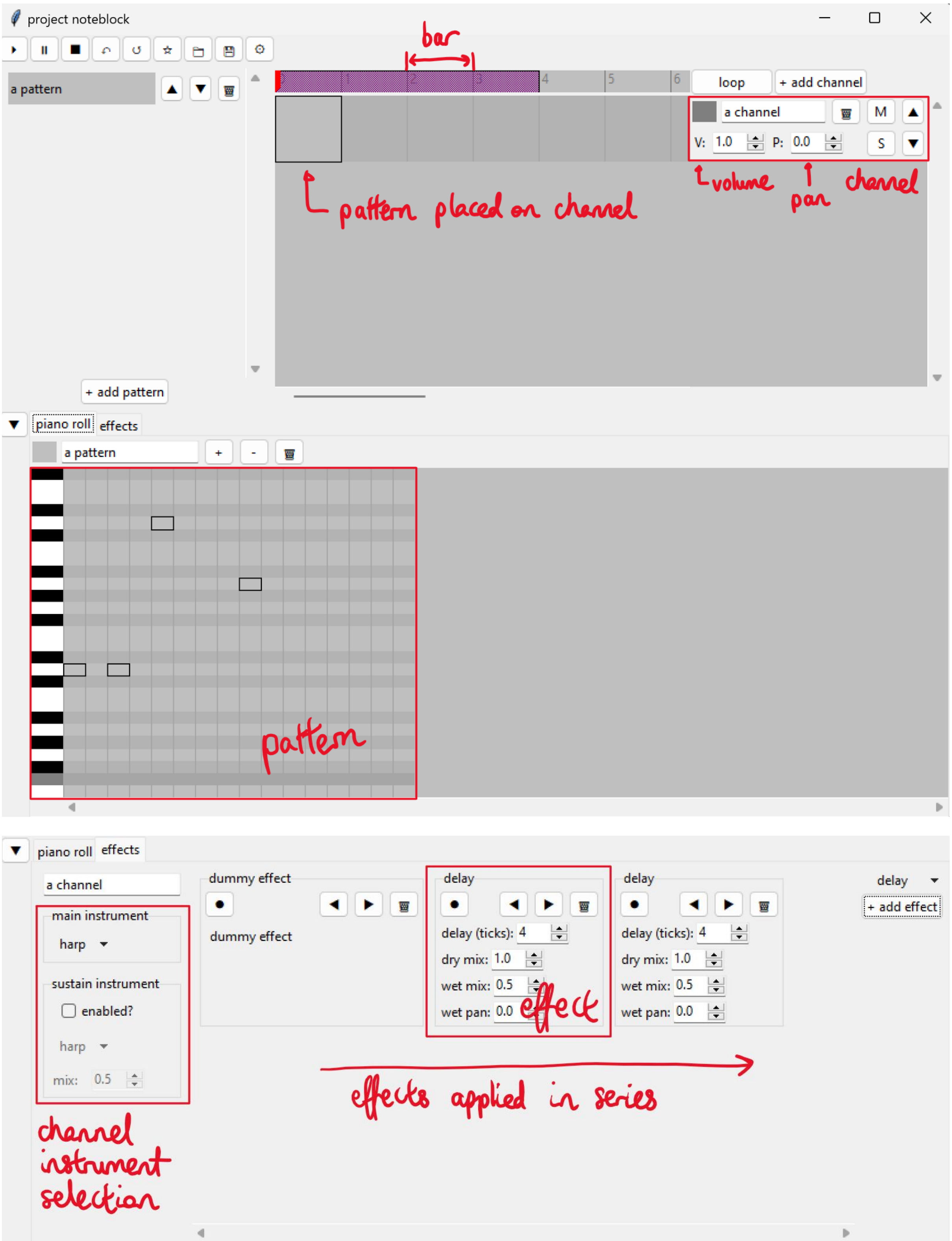
First, a discussion of the various abstract components of a song is called for; these components generally directly correspond to concrete objects in the data structure, as will be explained later.

A **pattern** is a sequence of notes to be played over time. For the sake of simplicity and performance, all patterns in a song are the same length in ticks. This can be quite creatively limiting, so I added an option to change the length of all patterns in a song, with an algorithm to stretch them. This system of having patterns independent of the arrangement of the song is based on that of FL Studio, and allows for an efficient workflow, as editing a pattern will make changes to all of its occurrences in a song, over time and across multiple channels.

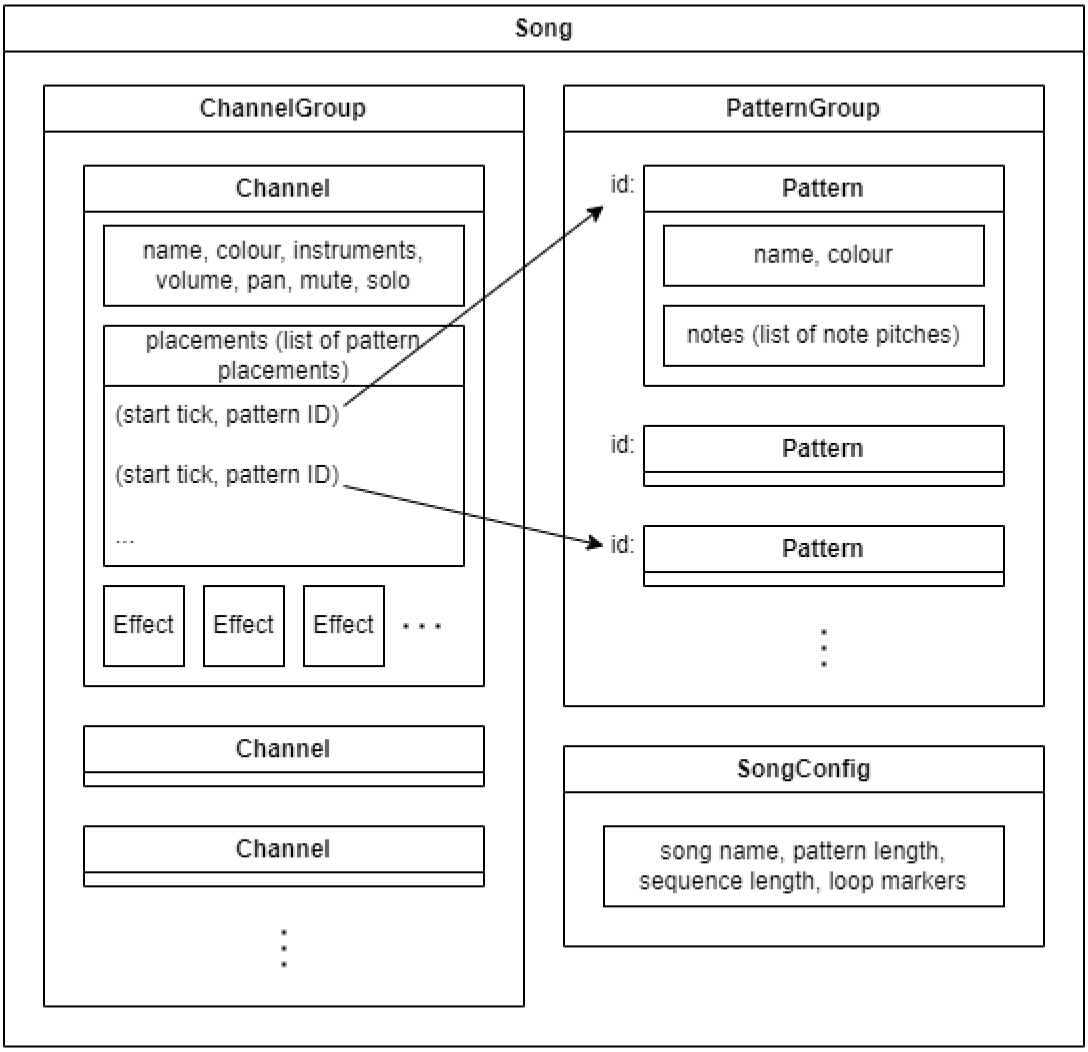
A **channel** essentially represents a voice in a song. It can be instructed to play given patterns at different points in time in a song, using various instruments available in Minecraft, and it can be positioned in the stereo field using options for volume and pan. The placement of patterns on given channels at given points in time is implemented in a manner similar to early versions of FL Studio, using the concept of **bars**: a bar is a unit of time, equal in length to a pattern, and each bar can contain a pattern. This means that patterns cannot be placed offbeat, but makes their placements easier to edit in the user interface, as well as more efficient to store and to process during playback.

Additionally, a channel can contain any number of **effects**: components that modify the notes being played on a channel in real time. For instance, I have implemented a “delay” effect, which uses a circular queue to store notes being played on a channel and repeats them at a lower volume after a fixed delay in ticks, creating an echo-like effect. Effects are applied on a channel in series and can thus be ordered in different ways to yield different results.

The following annotated screenshots of my program illustrate some of these concepts as they appear in the user interface.

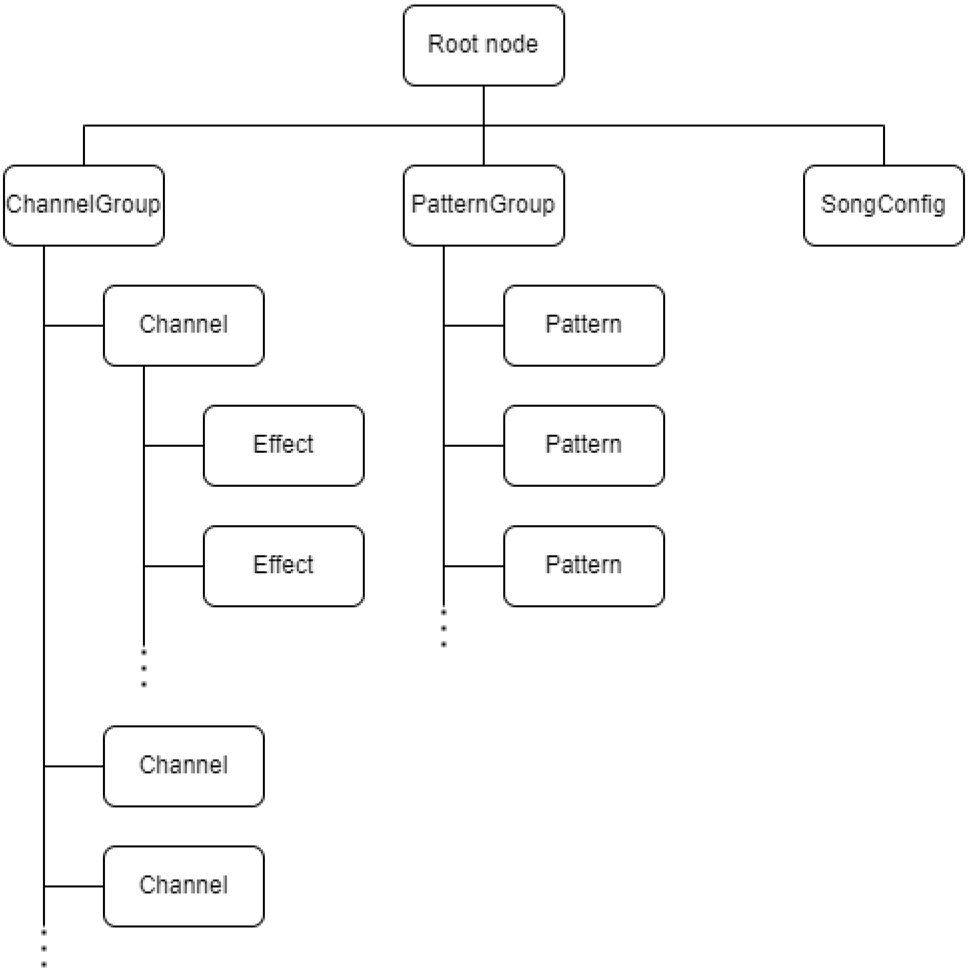


It can be observed that the various building blocks of a song naturally conform to a hierarchical structure: a song contains patterns and channels, and channels contain effects. The diagram below shows this structure in more detail.



As can be seen in the diagram, channels and patterns can have various attributes, such as name and colour, and both channels and patterns are stored within special group objects in the structure. The song object contains these groups, as well as general data about the song in a separate object. Channels can access the patterns that they are instructed to play via their unique ID within the PatternGroup object: this will be explained in more detail later in the document.

As this is a hierarchy of objects, it makes sense to use a tree as the actual data structure. The diagram below shows how the hierarchy is translated into such a tree.



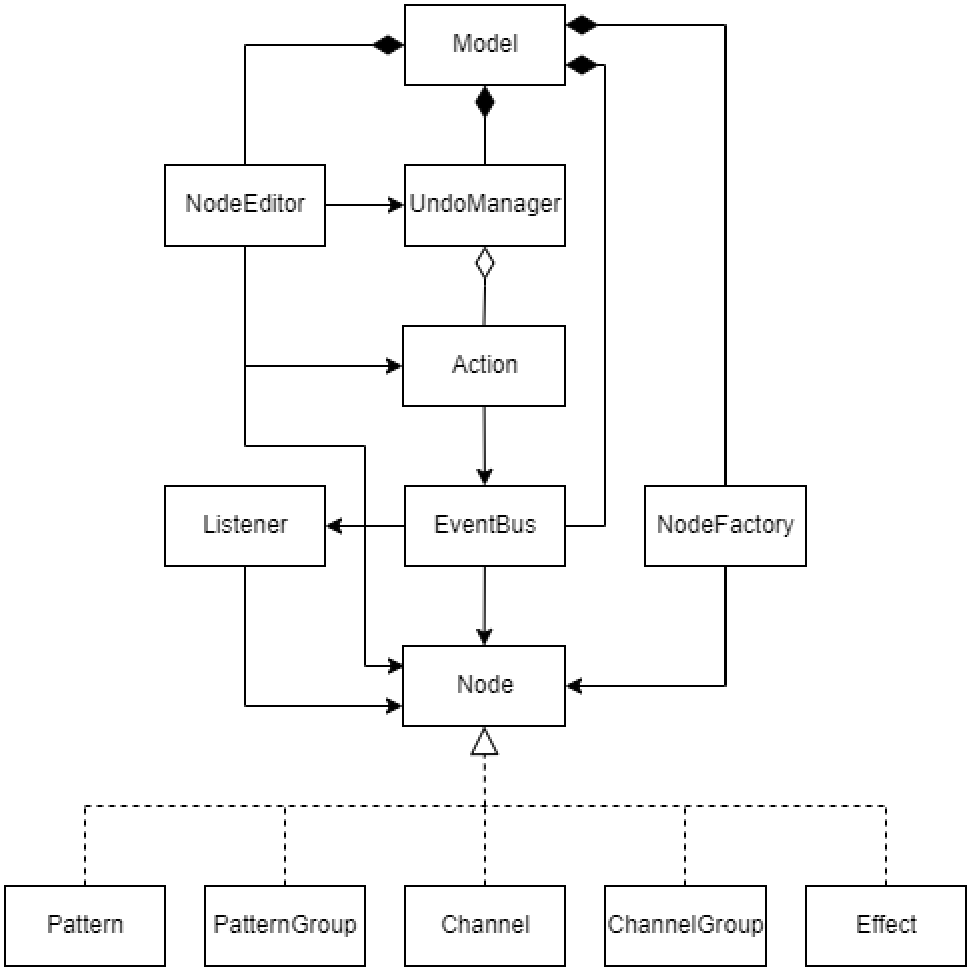
Model, UI, Playback

The classes and objects in my project can be broadly divided into three groups:

- The **model**, which contains the actual hierarchy of song objects as well as all the classes needed to support complex manipulation of those objects;
- The **UI**, which reads from the model to determine what to display, and is also able to write to the model;
- The **playback**, which reads from the model and determines how to turn it into audio data in real time.

Each of these components is explained in further detail below.

Data model



Designing and implementing the base class, the **node**, from which all components of my data model were to be derived, as well as the system of other classes built around it, was one of the major challenges of my project. I had several requirements for this class:

- **A tree structure**
  My data model contained relationships of components containing other components, for instance a song containing channels and patterns, or a channel containing effects. I decided that a tree structure was appropriate for this, due to its flexibility in which objects could own which other objects, while also maintaining control over every part of the tree in a way that was not possible with simpler data structures such as lists. Hence, a node contains references to its parent and its children, and I wrote methods for adding, removing and changing the order of children within a parent.

- **Support for undo/redo operations**
  A key feature that I wanted is the ability to undo and redo edits to a song. To achieve this, I used a behavioural design pattern commonly known as the "command" pattern. I encapsulated changes to a node as **actions**: objects that also have methods for reversing the action they perform. I then wrote an **undo manager**, which stores these actions and has methods for going back and forth in a history of edits. The last piece of this puzzle was a **node editor** object, which provides a wide variety of methods for editing a tree of nodes.

Under the hood, it creates action objects that can perform the requested edits, and passes them to the undo manager.
As well as a parent and children, each node may have some attributes that need to be saved and tracked by the edit history, and others that should not be. To achieve this distinction, I gave each node a dictionary of **properties**, which are saved and tracked, and anything that was not to be saved (**state**) I would store as normal attributes of the node object.

- **A way to listen to changes in the tree**
  I needed a way to notify other objects whenever an edit was made to the tree. This is primarily needed for GUI components, as they display representations of parts of the tree in real time. Updating them constantly would quickly cause performance issues, especially with a slower language like Python. Having GUI components only update when they are used to make an edit would not work either, as multiple parts of the GUI could depend on the same part of the tree, not to mention that I also needed undo and redo operations to trigger changes in the GUI.
  I solved this with a pair of classes: the **event bus** and the **listener**. The event bus stores a list of listeners (objects that inherit from the listener class) and has methods for adding and removing them. Both classes have a series of methods with the same names as each other: when one of these methods (also known as an **event**) is called on the event bus, it calls that method for all of its listeners. By including methods which represent changes to the tree and getting all action objects to call their corresponding events, the result was that all edits made through actions, including undo and redo operations, would notify listeners.

- **A way to serialize and deserialize the tree**
  Another essential feature of my project was the ability to save and load songs – that is, the actual structure of channels, patterns, effects etc., not just exported audio or schematics. I quickly settled on JSON for the file format, for multiple reasons:
    - It was a **safe, secure format**: I originally considered directly serializing the objects themselves into a binary format, for example using Python's built-in `pickle` module, but this would make arbitrary code execution exploits possible.
    - It was **unlikely to break**, even with significant changes to the format. I also considered manually creating compact binary representations of only the necessary parts of nodes in my own code, which would be more secure than pickling them; however, this would mean that as soon as I added even one more property to one node subclass, compatibility would be broken with earlier saved files. JSON is less compact as it stores everything in a text format, and includes the names of attributes as well as their values, but it meant that I was free to add properties to nodes as I pleased, and in the end the file sizes were still on the order of kilobytes.
    - It was **human-readable** and even editable by hand, making debugging much easier.
    - And lastly, Python already has a **built-in library** for reading and writing JSON, so I only had to concern myself with converting to and from a Python dictionary format.

  I wrote code for converting a node into a dictionary, which also recursively converts a node's children. Having a dictionary of a node's properties was useful here, as I could simply copy that dictionary in, and not worry about unintentionally saving any attributes to do with the state of the object.
  Then, when it came to converting a dictionary back into a node, I needed a way of storing information about the class of a node, and restoring the dictionary representations to the correct classes. I solved this by having nodes save their own class names as an attribute of

the dictionary format, and writing a **node factory** class, which was concerned with creating nodes from dictionaries. It looks up the class name in a dictionary which maps the names of classes to the actual classes, and initializes node objects from those classes. This is an inherently secure approach, as it can only create node objects from classes which I explicitly allow in code.

The last class in the data model that I have yet to explain is the one at the top, confusingly named **“model”**. In hindsight, a name such as “coordinator” or “driver” would probably have been more suitable; this is because the model class’s job is to instantiate all the other components, deal with loading song files into the data structure and saving song files from it, and provide methods which can make changes to a large number of nodes at once, for example changing the length of every pattern in a song.

Nodes and their children

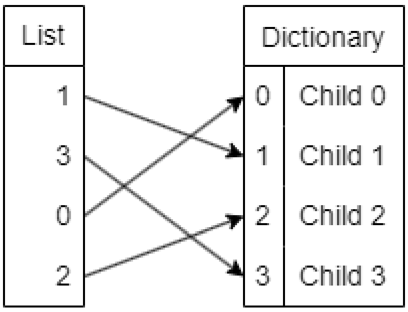
*See **node.py** (p.65)*

I needed to be able to specify and modify the order of children within a parent node. This is because the order of effects in a channel does influence its output, and additionally the ability to user-define the order of channels in a song is a feature I definitely wanted.

However, I also needed to be able to assign each child of a node its own unique ID, which would remain unchanged when reordering children. This is because channels had to refer to specific patterns, and direct references to their objects would not be preserved when serializing the tree.

This meant that Python’s built-in data structures were insufficient for storing a node’s children on their own, as lists do not have an efficient way to get items by ID, while dictionaries cannot be reordered.

I solved this by storing a list of numerical IDs in each node which are the keys to a dictionary of its children. Reordering the children of a node only affects the list, while adding and removing children are operations that modify both the list and dictionary. The diagram below illustrates how this system works.



If the children of a node are to be accessed in order, the list of IDs is iterated over and the child corresponding to each ID looked up in the dictionary and returned. Meanwhile, if a particular child of a node is to be accessed, its ID can be looked up in the dictionary directly, and since the ID of a child node does not change when children are reordered, this will always access the same child.

Actions and the undo manager

*See **node_actions.py** (p.67), **undo_manager.py** (p.92)*

`Action` and `UndoManager` are a pair of classes that together provide the ability to undo and redo edits made in the song editor, by maintaining and traversing a timeline of reversible actions.

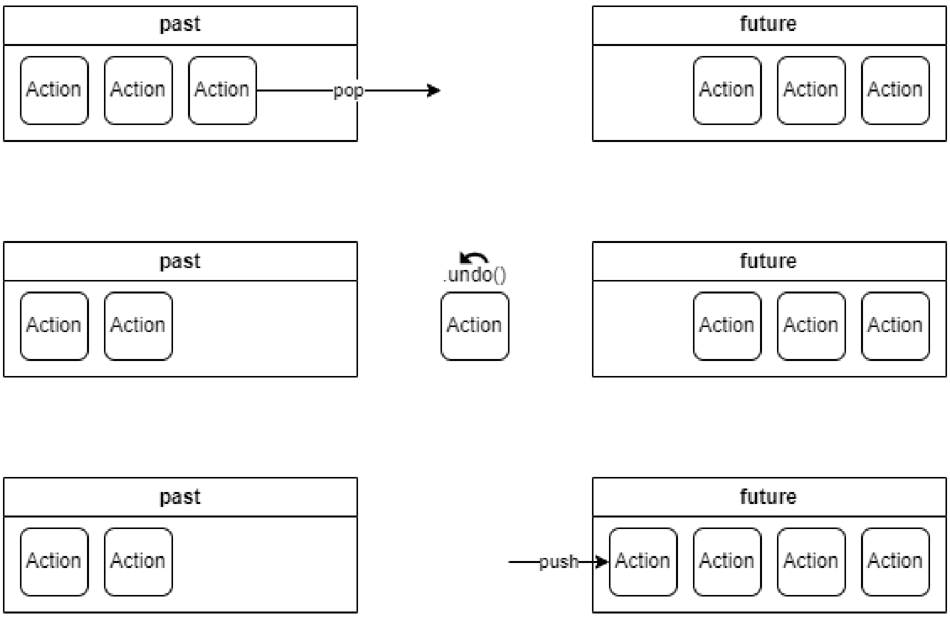
`Action` is an abstract class that encapsulates an action. It has two methods:

- `.perform()`, which is for doing the action; and
- `.undo()`, which is for reversing that action.

The operations that I need to be able to perform and reverse on a tree of nodes are setting a property on a node, adding a child to a node, removing a child from a node, and moving a child within the order of children of a node. I implemented each of these operations as a subclass of the `Action` class, with appropriate ways of reversing them. For example, adding and removing children of a node are the reverse actions of each other, and an `Action` object can reverse setting a property by storing the old value of that property upon initialization, and setting the property's value back to that old value when its `.undo()` method is called. Then, by initializing these subclasses with parameters that supply the nodes and values involved in the operations, the program can handle every edit made to a tree of nodes as an `Action` object.

`UndoManager` is the class that actually handles the timeline. It contains two stacks of `Action` objects, one that represents actions that occurred in the past and can be reversed, and one that represents actions that occurred in the future and can be performed again. To undo the last performed action, the undo manager pops an `Action` from the "past" stack, calls its `.undo()` method, and pushes it onto the "future" stack. Similarly, to redo the last undone action, it pops an `Action` from the "future" stack, calls its `.perform()` method, and pushes it onto the "past" stack. In this way, the chronological sequence of actions is preserved through undo and redo operations. The logic for doing a new action not yet on the timeline is slightly different: as well as calling the `Action`'s `.perform()` method and pushing it onto the "past" stack, the undo manager clears the "future" stack, as the timeline of actions has diverged from its previous state and those future actions no longer apply.

The diagram below demonstrates undoing an action as described above.



Listeners and the event bus

*See **events.py** (p.56)*

I described the system of the event bus and listeners in a previous section, but it is worth elaborating on. When the program is run, only one event bus is created, and it acts as a universal message carrier across all components of the program. However, listeners are generally UI components (except for some parts of the playback system), as they are the parts of the program which need to be updated in response to changes to the model. Listeners add themselves to the event bus on initialization and remove themselves on deletion.

Unfortunately, using a universal event bus means that any edit to the tree notifies every registered listener, whether they are dependent on that part of the tree or not. In fact, the first solution I tried involved each node having its own connections to listeners. However, I could not figure out a way to easily and reliably maintain or re-establish these connections when parts of the tree were removed. Ultimately, I just made sure to minimise the performance impact of the event bus system by making sure that the very first thing that all listeners did when receiving events was filtering out any that they were not interested in.

The different events that I included in the event bus are:

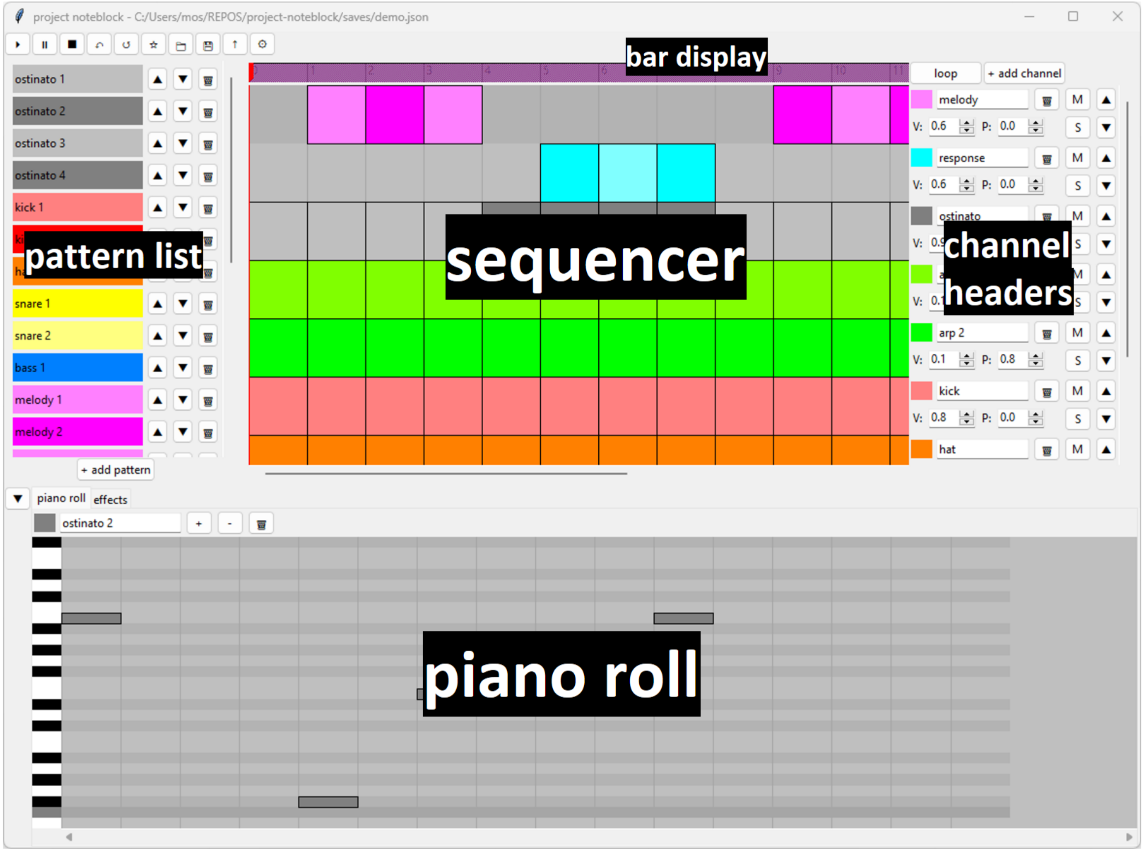
- `node_property_set(node: Node, key, old_value, new_value)`
- `node_child_added(parent: Node, child: Node, id: int, index: int)`
- `node_child_removed(parent: Node, child: Node, id: int, index: int)`
- `node_child_moved(parent: Node, old_index: int, new_index: int)`
  These self-descriptive events correspond to changes to nodes. They are triggered by `Action` objects after they have made said changes to nodes, so that UI components can respond to these changes as required.

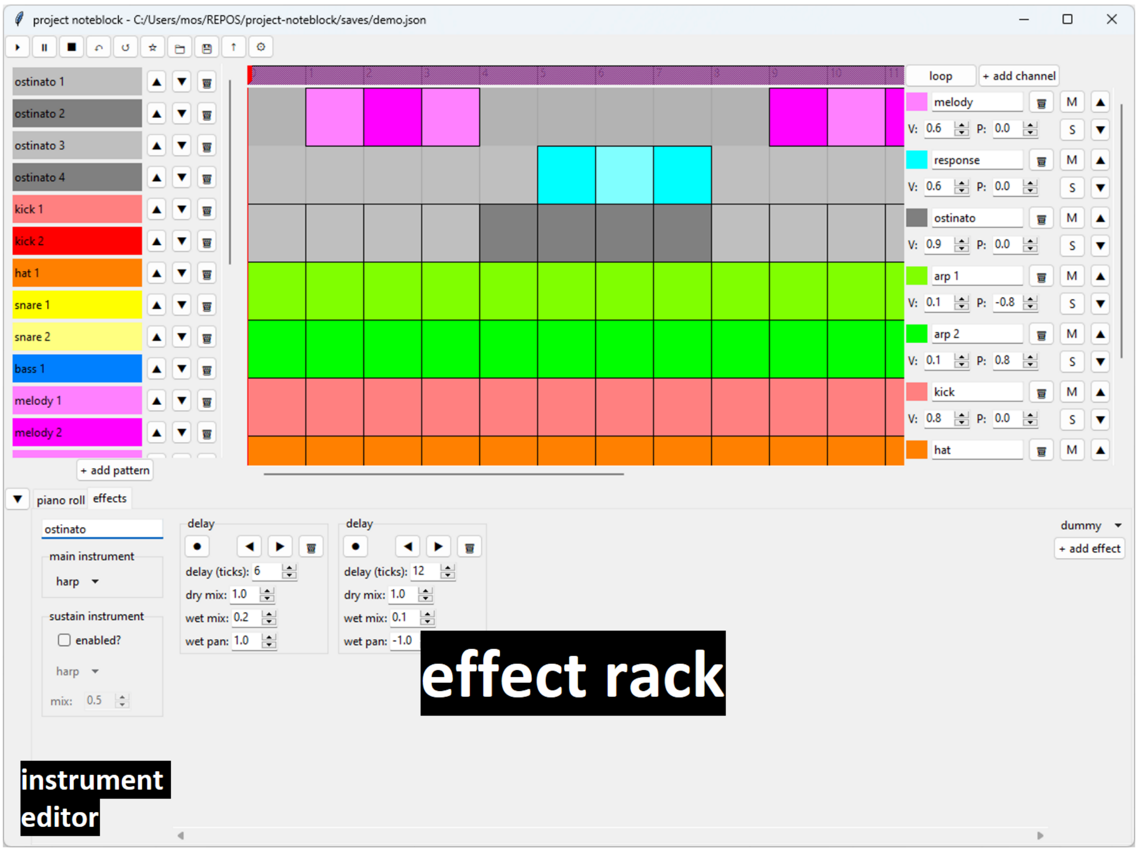
- `node_selected(node: Node)`
  This event is triggered when a UI component that directly corresponds to a particular node is clicked by the user, so that other UI components can switch to displaying info specific to that node. For example, when a channel is selected in the sequencer, the instrument/FX panel displays the instrument and effect rack of that channel.

- `bar_selected(self, bar: int)`
  This event is triggered by the sequencer when a particular bar is selected in the sequencer, so that the playback logic knows that it should start playing the song from that bar.

- `reset_ui(self):`
  This is called by the program's central `Model` object when a song is loaded from a file, so that all UI components know to delete any references they have to nodes from the previous song. This is because loading a song from a file creates a new tree of nodes from scratch and disposes of the previous one.

User interface





In the user interface of my program, there are a number of different components whose designs are worth discussing.

The **sequencer** (see **sequencer.py** on p.86) is the main view of the song in the program: it shows all of the song's channels, laid out vertically, and all of its bars, laid out horizontally, with their intersections forming a grid in which patterns can be placed. As a result, it is scrollable in both directions. To the right are the **channel headers** (see **channel_header.py** on p.42), which are lined up with their corresponding channels in the sequencer; they allow the user to edit the properties of those channels. Above the sequencer is the **bar display** (see **bar_display.py** on p.37), which shows bar numbers aligned with the sequencer, and allows the user to set playback start and loop points. The channel headers only scroll vertically with the sequencer, while the bar display only scrolls horizontally; in this way, they are always visible even as the sequencer is scrolled in all directions.

Clicking in the sequencer will select the relevant channel, bringing up its instrument editor and effect rack; the relevant bar, setting the playback start point; and the relevant pattern (if one has been placed at the point clicked), bringing it up in the piano roll. This makes using the sequencer highly interactive and intuitive.

To the left of the sequencer is the **pattern list** (see **pattern_list.py** on p.72): this shows all of the song's patterns, with their set colours, in a scrollable list. Again, these patterns can be clicked to bring them up in the piano roll; the pattern list also allows the user to add/remove and reorder patterns. To place a pattern into the song sequence, the user simply has to drag it from the pattern list to the desired channel/bar intersection in the sequencer.

Below the sequencer and pattern list, there are two tabs, which can be swapped or completely hidden as desired.

The first tab is the **piano roll** (see **piano_roll.py** on p.78): this provides a MIDI-like gridded display of a pattern which can be edited. The vertical axis represents pitch, and the horizontal axis is time, with the pattern's notes displayed at appropriate coordinates. The piano roll can be zoomed in/out, scrolled, and clicked to add or remove notes.

The second tab is the **instrument editor** (see **instrument_editor.py** on p.57) and **effect rack** (see **effect_rack.py** on p.52). The instrument editor allows the user to edit the instruments with which a channel plays its patterns, while the effect rack is a scrollable display of the channel's effects, in order, where they can be added/removed, reordered, enabled/disabled and modified.

Finally, at the top of the interface are a row of buttons which accomplish general tasks (see **top_frame.py** on p.90), allowing the user to play/pause/stop playback, undo/redo changes, load/save songs or create new ones, and edit a song's settings in a pop-up dialog.

I will discuss the implementation of the user interface only briefly, as the process of writing it consisted mostly of wrestling with `tkinter`, the GUI library, in order to create the layouts of the various UI components in code to a satisfactory degree of appearance. However, all UI components in the program follow the same general structure:

- They are all implemented as classes, subclassed from built-in `tkinter` UI components.
- They have two primary methods, `init_ui()` and `update_ui()`.
    - `init_ui()` is run upon initialization and creates all the sub-components contained within the component. It also registers the component as a listener to the event bus in order to react to events such as changes in the data model or selection of song

objects, as well as putting callbacks in place with the `tkinter` event loop in order to respond to actions such as mouse events.

  - `update_ui()` is called indirectly through the event bus: the UI component overrides specific listener methods to call `update_ui()` depending on what events it wants to react to. In particular, for data model related events, the UI component will generally read all the parts of the data model relevant to itself and update its appearance accordingly.
  - A good example of these methods in practice can be found in **channel_header.py** (p.42). The `init_ui()` method creates a variety of UI components, placing them appropriately within itself, and the `update_ui()` updates all of these components based on values from song objects.
- Some UI components may have additional methods that update specific parts of the component, called via specific events. For example, the sequencer (most importantly **placement_display.py** on p.81) has separate methods for redrawing the background grid, pattern placements, and playback start marker, which are triggered by different events; redrawing the background grid only needs to happen if channels or bars are added or removed, while redrawing the playback start marker is triggered by the `bar_selected` event, as selecting a bar sets the place from which playback begins.

I structured the user interface components hierarchically, with components containing other components inside of them, in order to make them more flexible and easily reconfigurable. For example, the buttons in the top row of the interface are instances of `tkinter`'s `Button` class, and they are all contained within another component, called `TopFrame`, which itself is subclassed from a generic container component from `tkinter` called `Frame`. The channel headers in the sequencer are instances of the `ChannelHeader` class, also subclassed from `Frame`, and this class creates instances of `tkinter` component classes such as `Button` for the buttons, `Entry` for the text input, and `Spinbox` for the numerical inputs.

Real-time playback and the loop hijacker

*See **loop_hijacker.py** (p.61)*

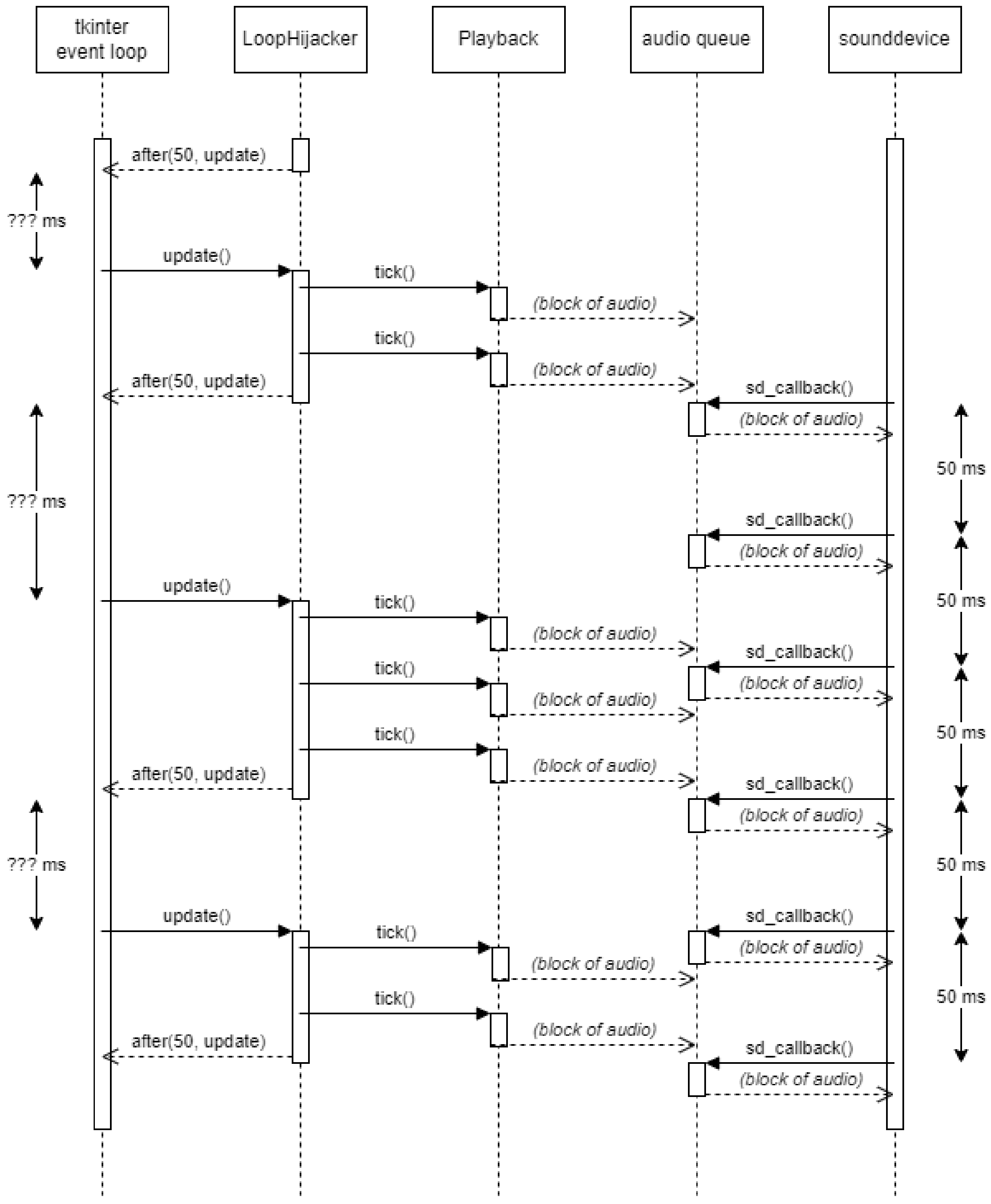
Smooth, stutter-free real-time audio playback is a key part of my project, as it is important to the music making process to be able to hear what you have made. However, Python certainly does not make it easy. The only library I could find for streamed audio output in Python was `python-sounddevice`, which runs itself on a separate thread, regularly calling a callback function in which you are expected to give it a block of audio in the form of a `numpy` array, which it will then play on the output device you have selected. Minecraft runs at 20 ticks per second, so to keep things simple, I generate audio in blocks of 1/20th of a second in length, once every 1/20th of a second.

However, doing all of the audio processing within this callback would be a bad idea, both because the library expects the callback to return as quickly as possible, and also because the playback mechanism needs to access the song node tree and retrieve the notes to be played from it, and doing this from a separate thread means that it could potentially try to read from the node tree while it is in an invalid state. So, I instead have the playback logic run on the main thread and place generated blocks of audio on a thread-safe queue (part of Python's standard library) which the output callback can then read from.

But this approach has its own issues, in particular that the playback logic now has to work around the GUI library, `tkinter`, which continually runs an event loop on the main thread. The main window class has a method, `.after(delay_ms, callback)`, which lets you specify a callback that the event loop will try to run after a given time in milliseconds, but this gives no guarantee that your callback will be run after exactly that length of time. In fact, it will usually be significantly delayed, as other processes in the event loop, such as updating the on-screen GUI, take priority over it. This is disastrous for real-time audio, because if the audio cannot definitely be generated ahead of time, then audio glitches will occur.

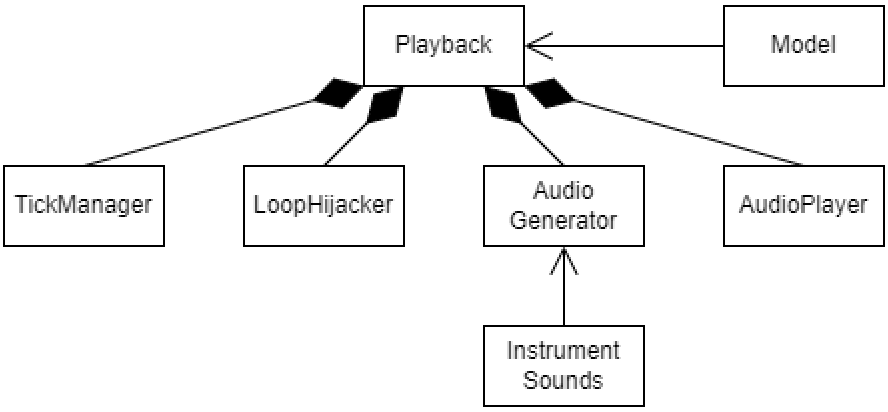
This is where the `LoopHijacker` class comes in. This class sits as a layer in between the `tkinter` event loop and the playback logic and makes sure that the latter is run enough times, ahead of time, at all times. It has a method, `.update()`, which compares the current real-time tick, calculated from the system clock, with the last processed tick, and runs the playback logic repeatedly until they are equal. Additionally, it has a "lookahead" feature, which makes it so that the playback logic is actually run a specified number of ticks into the future compared to the current tick, so that audio generation never falls behind.

The following diagram illustrates this process. Observe that the `sounddevice` callback is always able to pull a block of audio from the audio queue, even when the `tkinter` event loop is very inconsistent at calling our code.


tkinter event loop
LoopHijacker
Playback
audio queue
sounddevice
after(50, update)
??? ms
update()
tick()
(block of audio)
sd_callback()
50 ms

Playback

*See **playback.py** (p.85)*



Audio playback is handled overall by the `Playback` class. This class creates a `LoopHijacker` and uses it to hook itself into the `tkinter` event loop, from where it uses a variety of other classes in tandem to generate and play audio every tick. The playback logic itself can be broadly divided into two parts:

- **Note processing**, in which the objects in the model which make up the song data are instructed to produce one tick's worth of notes; and
- **Audio generation**, in which the `AudioGenerator` class turns these notes into a block of audio, which is then sent to a playback device via the `AudioPlayer` class.

The reason for splitting up the playback logic in this fashion is that while note processing logic is intrinsic to the instances of `Node` subclasses which make up the song data, such as patterns, channels and effects, audio generation can be considered a separate component which is optionally attached at the end. The lists of notes could alternatively be fed to code which turns them into a Minecraft in-game structure, or MIDI files to be imported into other music software. While I have as yet not created these components, separating note processing and audio generation makes it easier for them to be written and integrated into the program.

Note processing and song objects

This is an opportune point at which to explain the workings of the song objects (`Node` subclasses) in detail. However, I will first explain the `Note` class, as while it is not a song object, it is still integral to the whole process.

Note

*See **note.py** (p.71)*

Note
+ instrument: int + pitch: int + volume: float + pan: float
+ apply_volume_and_pan(volume: float, pan: float) -> Note

The `Note` is the primitive unit of data handled by and exchanged between all objects which process notes. It mostly functions more as a struct than as a class, but Python does not have structs, so I made it as a class instead.

The **instrument** attribute is an integer between 0 and 15 inclusive which corresponds to a note block instrument, and the **pitch** attribute is an integer between 0 and 24 inclusive which maps to a pitch within a note block's two octave range. These both directly correspond to attributes which Minecraft itself assigns to note blocks in-game.

Additionally, the **volume** attribute indicates how loud a note is to be played, with a range from 0 (silent) to 1 (full volume), and the **pan** attribute indicates where a note is to be placed in the stereo field, with a range from -1 (far left) to 1 (far right).

Lastly, the `apply_volume_and_pan` method is for easily adjusting a note's volume and pan, with parameters specifying a change from its current state as opposed to entirely new values for the attributes. However, instead of changing the object's own attributes it actually creates a new `Note` with the modified values. This is because some effects may wish to keep references to `Note` objects across several ticks, and so those objects should not be modified by other components. To calculate the new volume and pan, the note's volume is multiplied by the given volume, and the given pan is added to the note's pan, after which both attributes are clamped to within their respective ranges.

Pattern

*See **pattern.py** (p.72)*

Pattern (Node)
+ name: str + colour: str + notes: list[int]
+ get_notes(pat_tick: int) -> list[int]

The `Pattern` is a container for a sequence of notes. However, these are not stored or output as `Note` objects, which are first created in the `Channel`. Instead, a `Pattern` object contains a list of pitches, stored as integers, which represent a series of notes over time, one every tick. In addition to the standard 0 to 24 range, these pitches can take two other special values: -1 (meaning no note) and -2 (meaning sustain release), which will be explained later.

The `get_notes` method takes the tick within the pattern as a parameter and returns the corresponding pitch at that time. This returns a list of integers instead of a single integer as I was at one point considering implementing polyphony within patterns, and wanted to leave myself that option. The **name** and **colour** attributes are primarily for use within the GUI so that they can be displayed to and changed by the user to aid with song organisation.

Patterns are stored as children of a `PatternGroup` object, which is also subclassed from `Node`; this means that they can be reordered freely while also being referenceable by an unchanging ID within the pattern group.

Note that for subclasses of `Node`, any public attributes shown in the class representation are actually stored as properties as described in the "data model" section, so that they can be loaded and saved, undone and redone, trigger GUI updates on change, and so on.

Effect

*See **effect.py** (p.48)*

<<interface>> **Effect (Node)**
+ enabled: bool
+ tick(notes: list[Note], mono_tick: int) -> list[Note] - process_notes(notes: list[Note], mono_tick: int) -> list[Note]

The `Effect` is an interface for note processors which modify notes as they pass through a channel. Its `tick` method has a list of `Note` objects as both input and output, so effects can be chained together and applied in series on a channel. However, this method actually just checks if the **enabled** flag of the effect is true, and calls the `process_notes` method if it is. This latter method is what implementations of the `Effect` interface can override in order to provide their own functionality. In a similar vein to `Note` objects being immutable, these lists of `Note` objects are not actually modified by effects; the effects instead create new lists to be returned as output.

Channel

*See **channel.py** (p.41)*

Channel (Node)
+ name: str + colour: str + main_instrument: int + sustain_enabled: bool + sustain_instrument: int + sustain_mix: float + volume: float + pan: float + mute: bool + solo: bool + placements: list[int] - pattern_group: PatternGroup - sustained_note: int
+ tick(mono_tick: int, sequence_enabled: bool, bar_number: int, pat_tick: int) -> list[Note] - convert_numbers_to_notes(note_numbers: list[int]) -> list[Note]

The `Channel` is where most of the heavy lifting of note processing is done. This class has the tasks of:

- referencing and using patterns based on how they have been sequenced in the song
- creating `Note` objects
- processing instruments and sustain
- applying effects
- applying a final volume and pan setting

Effects are not shown among the properties of `Channel`; they are instead stored as its children within the node tree structure.

Note that the `tick` method, which does all the things listed above, has a number of arguments, which together make up information about the current tick being processed.

- **mono_tick**, short for “monotonic tick”, is a tick counter which is initialized to 0 when the playback engine first starts and is always incremented every tick.
- **sequence_enabled** is a flag indicating if the song is currently playing. The playback engine always ticks channels regardless, both so that effects can continue to tick even when song playback has been paused, and so that the program does not have to deal with frequently stopping and starting the hardware audio output. Hence, this flag tells the channel whether or not it should also pull notes from patterns in the song sequence on the current tick.
- **bar_number** and **pat_tick** together describe the current tick within the song sequence, with the latter referring to the current tick within the specified bar.

The real-time tick and sequence tick are both supplied as arguments as they have different uses: the former is used by effects, allowing them to operate in real-time (for example, a delay effect should always delay notes by the given number of ticks, regardless of whether the song has been looped back or paused), whereas the latter is used by channels to retrieve the corresponding patterns in the sequence.

The “song sequence” referred to up until now is stored in the **placements** property of each channel. This is a list of integers, in which each successive item in the list refers to the pattern to be played by the channel in that bar (the first item refers to the pattern in the first bar, etc.). The integers themselves actually correspond to the IDs of patterns stored in the song’s `PatternGroup` node as its children, with -1 meaning that no pattern is to be played in that bar. Hence, channels store a reference to the song’s pattern group, so that they can look up its children (the song’s patterns).

Putting this together, when a channel’s `tick` method is called, it first:

- Gets the current bar from the method's parameters
- Uses it as an index to its **placements** list to get the current pattern ID
- Uses that as a key to the pattern group's dictionary of children to get the current pattern
- Calls the pattern's `get_notes` method, with the parameter being the current tick within the bar to be played, to get a list of integers representing pitches of notes.

Next, the channel feeds this list of pitches to its own `convert_numbers_to_notes` method. This is responsible for turning the pitches into `Note` objects, but is more than just a simple converter, as it also handles "sustain notes" – notes which are held down for a longer time. Minecraft note blocks do not provide a built-in way to sustain played notes, and no other Minecraft music editors have implemented this as a feature, so it was something I was certain I wanted.

The way I did this was to play a note repeatedly, every tick, as an approximation to sustaining the note. This is what the **sustained_note** attribute is for: it holds the pitch to be sustained, or None if no note is to be sustained. Then, when pitches are passed to the `convert_numbers_to_notes` method, the **sustained_note** attribute is set to the last given pitch, to continue sustaining that one until the next pattern-supplied pitch. The purpose of the "sustain release" value of -2 in a pattern is to tell the channel to set **sustained_note** to None and stop sustaining any notes. If no pitches are received from the pattern on a given tick, then **sustained_note** is used as the pitch to generate a `Note` object from instead.

So, the next steps in the `tick` method are:

- Convert the list of pitches into a list of `Note` objects, including handling sustained notes
- Feed the list of `Note` objects through each of the `Effect` objects which are the channel's children, taking the returned list from each effect as the input for the next
- Apply a final balance at the end based on the **volume** and **pan** properties, using the `Note` objects' `apply_volume_and_pan` method
- Finally, return the resulting list of notes

ChannelGroup

*See **channel_group.py** (p.42)*

ChannelGroup (Node)
+ tick(mono_tick: int, sequence_enabled: bool, bar_number: int, pat_tick: int) -> list[Note]

As the `PatternGroup` class holds a song's patterns, so the `ChannelGroup` class holds a song's channels. However, there is some more interesting logic happening here, especially regarding the **mute** and **solo** properties of the `Channel` class, which I did not explain above.

Mute and solo are concepts common to practically every DAW, and they work as follows: if a channel is muted, then it does not play audio (notes in this case), and if any channels are soloed, only those channels play, and all others are muted. The corresponding flags in the `Channel` class are in fact processed in the `ChannelGroup` class, as knowledge of the status of all channels is required to implement soloing correctly.

The procedure performed by the `ChannelGroup` class in its `tick` method to implement this functionality is best represented with pseudocode, and is as follows:

```
notes = []
solo_active = true if any of my child channels have the
      solo flag set to true, otherwise false
```

```
for each of my child channels:
      channel_notes = list of notes from ticking this channel
      if this channel is soloed,
                  or (it is not muted and solo_active is false):
            append channel_notes to notes
return notes
```

TickManager

*See **tick_manager.py** (p.89)*

TickManager
- mono_tick: int - sequence_enabled: bool - bar_number: int - pat_tick: int
+ set_tick(bar_number: int, pat_tick: int) + next_tick() -> tuple[int, bool, int, int] - increment_tick() - justify_tick()

It can be seen above that the `Channel` and `ChannelGroup` classes both have a method, `tick`, to generate notes for each tick, and which takes a variety of information about the current tick as its parameters. To play through a song, this information must be generated somewhere, and this is what the `TickManager` class is for. All the parameters which comprise tick information are present in this class as private attributes, and these form a counter that is incremented after each time it is requested by another object through the `next_tick` method. This method caches the current tick information, then calls the private `increment_tick` method before returning the cached information.

`increment_tick` is simple: it first increments the **mono_tick** (as this is always going up every tick whether the song is playing or not), then checks if **sequence_enabled** is true. If it is, then this means that the song is playing, and the current tick in the sequence should also be incremented. So, first **pat_tick** is incremented, then the other private method, `justify_tick`, is called.

This other method is responsible for making sure that the stored tick information corresponds to a valid pattern tick in a valid bar of the song, that the song's loop markers are respected if looping is enabled, and that song playback stops when the end of the song is reached. The algorithm for this is laid out in pseudocode below.

```
if pat_tick >= pattern length in ticks:
      pat_tick = 0
      bar_number = bar_number + 1
      if bar_number == end of loop, and looping is enabled:
            bar_number = start of loop
      if bar_number >= length of sequence in bars:
            sequence_enabled = False
```

Note that in this procedure, most of the variables checked (pattern length, bar length, loop start/end/enabled) are properties in a special node, `SongConfig`, which together with a song's `PatternGroup` and `ChannelGroup` nodes make up the root nodes of a song's data. These properties are stored separately to the patterns and channels of a song because they are intrinsic data about the song itself, and not any single pattern or channel.

Audio generation

After all note processing has been performed on a given tick, the resulting list of `Note` objects needs to be turned into a block of audio. This is the job of the `AudioGenerator` class, which will be explained shortly, but first it would be more appropriate to explain another class, `InstrumentSounds`, which is closely related.

InstrumentSounds

*See **instrument_sounds.py** (p.60)*

InstrumentSounds
- block_size: int - sounds: dict[int, list[np.ndarray]]
+ get_sound(instrument: int, pitch: int) -> np.ndarray

The purpose of the `InstrumentSounds` class is to generate and store an array of audio data for every instrument at every possible pitch, and provide a simple interface with which to access these arrays. There are 16 different note block instruments in Minecraft, and 25 different possible pitches for a note, so storing the generated audio data in RAM is not very memory intensive, only taking up about 50MB of memory in total, which is very acceptable.

For each instrument, I have a .wav audio file of the instrument's sound extracted from Minecraft, which is played at the root pitch, exactly in the middle of the instrument's range. However, this range extends an octave above and below the root pitch, so I needed to resample the audio data at different sample rates, which can be expressed as a ratio with the original sample rate of the audio file.

These ratios can be derived mathematically. To transpose audio down an octave, you sample it at double the original sample rate, then play this new audio data at the original sample rate. Similarly, to transpose audio up an octave, you sample it at half the original sample rate. To derive the ratios for all the pitches in between, twelve-tone equal temperament (the predominant tuning system of Western music) is used. An octave is split into 12 equally spaced semitones, and the ratio between semitones is constant. Hence, this ratio must be the 12th root of 2 (approximately 1.059463), and intervals of multiple semitones are derived by taking the corresponding power of this ratio.

Additionally, when sounds are loaded into memory, they are padded with extra silence at the end to bring their length in samples up to a multiple of the attribute **block_size**. This is so that the `AudioGenerator` class does not have to deal with adding together arrays of differing lengths when generating blocks of audio.

The data structure which holds all the audio data at the end is a dictionary, with keys corresponding to the IDs of the instruments. Each instrument's entry is a list of arrays of audio data, with the index in each list corresponding to the pitch of the audio sample. The `get_sound` method abstracts away this complex indexing process, simply taking arguments for instrument and pitch instead.

The following pseudocode illustrates the whole process of loading, generating and storing audio data, performed by the `InstrumentSounds` class upon initialization:

```
sounds = {}
pitch_ratios = []
for each pitch from 0 to 25:
      append 2 * (2 ^ -pitch/12) to pitch_ratios
for each instrument:
      sound_list = []
      read instrument audio from file
```

```
        for each ratio in pitch_ratios:
            resample audio with the given ratio
            pad_length = block_size –
                        (length of pitched audio % block_size)
            add (pad_length) samples of silence to end of audio
            append audio to sound_list
        add sound_list to sounds with instrument’s ID as the key
```

AudioGenerator

*See **audio_generator.py** (p.36)*

AudioGenerator
- block_size: int - sounds: InstrumentSounds - current_sounds: list[tuple[np.ndarray, int]]
+ tick(notes: list[Note]) -> np.ndarray - process_new_notes(notes: list[Note]) - tick_audio() -> np.ndarray - stereo_volumes(note: Note) -> np.ndarray

The `AudioGenerator` turns lists of notes into blocks of audio.

Note the **block_size** attribute, which determines how long a block of audio is in samples. As the entire playback logic is run 20 times a second, a block of audio is 50 milliseconds long (1/20th of a second) in order to make everything line up. In fact, an `AudioGenerator` object creates its own `InstrumentSounds` object (stored as the **sounds** attribute), and passes its own **block_size** to it.

More interesting is the **current_sounds** attribute, which the audio generation algorithm revolves around. This is a list of every sound currently playing, as well as how much of each sound has already been played, given as an index into the sound’s audio array. This is necessary because most of the sounds are more than one tick long, so their audio data has to be split up and played over multiple ticks. The task of the `tick_audio` method is to iterate over the list, cutting out the right slices of each audio array and summing them together into a block of audio. This can again be represented in pseudocode:

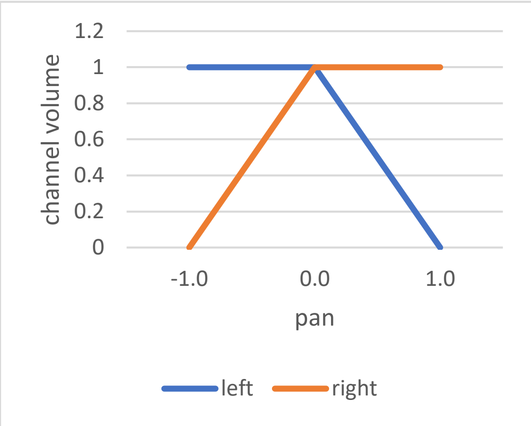
```
audio_block = new empty block of audio
new_current_sounds = []
for each sound and its start_index:
    end_index = start_index + block_size
    sound_slice = a slice of the sound array
                from start_index to end_index
    if end_index < length of sound:
        append sound and end_index to new_current_sounds
    add sound_slice to audio_block, element-wise
current_sounds = new_current_sounds
return audio_block
```

Mixing audio is accomplished in a number of different ways. Summing several sounds into one block of audio is easy, as demonstrated above: you simply add the arrays of audio together, element by element. Applying volume and pan to audio is slightly harder, and is actually done before a sound enters the **current_sounds** list.

An array of audio has two dimensions: the first is the number of samples, and the second is the number of channels. In this case the word “channel” has a different meaning from what I have been using it for above – essentially one channel corresponds to one speaker, so stereo audio is accomplished with two channels which map to the left and right speakers.

This means that an array of audio can be multiplied, element-wise, with a 2-element array containing the volumes for the left and right speakers, and these volumes will be applied to the left and right channels of the audio respectively. The left and right volumes can be calculated from the **volume** and **pan** properties of a `Note` object as following:

```
vol_left = volume * max(1 - pan, 1)
vol_right = volume * max(1 + pan, 1)
```



The way this calculation works can be seen in the graph on the left. The expression involving pan calculates volume multipliers for both channels that can go from 0.0 (silence) to 1.0 (full volume). It can be seen that a fully left-panned sound (pan = -1.0) is only played in the left channel, a fully right-panned sound (pan = 1.0) is only played in the right channel, a centre-panned sound (pan = 0.0) is played at full volume in both channels, and anywhere in between these points is linearly interpolated between these absolutes.

The `stereo_volumes` method performs this calculation for a given note, and is used by the `process_new_notes` method. This method takes a list of notes, uses each note's **instrument** and **pitch** properties (as parameters for the `InstrumentSounds` object) to get an audio array of the sound, and multiplies this array by the channel volumes calculated from the note's **volume** and **pan** properties, before finally appending the array to **current_sounds** with a start index of 0. This is how new notes are processed, and in fact the public `tick` method of this class, which receives the notes generated on each tick, just calls the two private methods.

Assorted algorithms

In this section, I will explain some algorithms present in my code which did not have an obvious place in the other sections of the design but are still worth examining in more detail.

Pattern stretching

*See **change_pattern_length()** in **model.py** (p.63)*

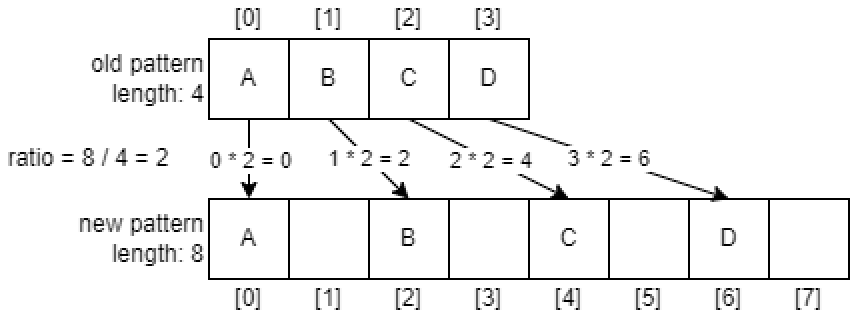
In a song, all patterns are the same length. This allows them to be placed cleanly into bars in the sequence, but also makes it difficult to change the effective BPM or time signature of a song. To rectify this, I wrote a method in the `Model` class which can change the length of all patterns in a song simultaneously, stretching the timing of existing notes in patterns relative to this length.

To accomplish this stretching, I designed an algorithm which, while simple, has some interesting subtleties of note. The pseudocode for the algorithm is as follows:

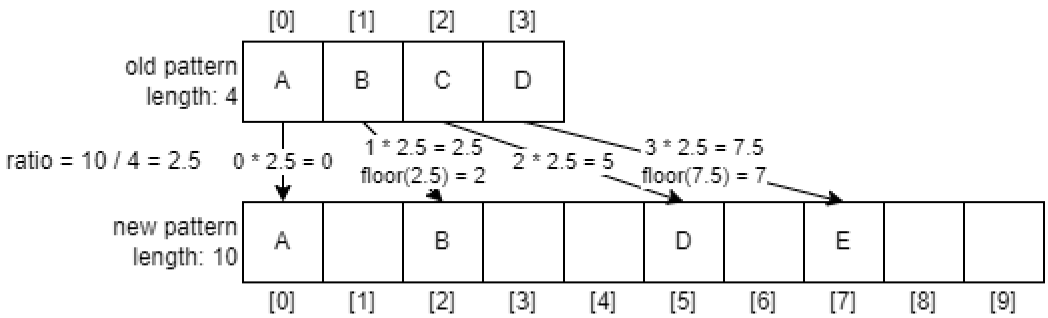
```
determine old_length and new_length of pattern
ratio = new_length / old_length
old_notes = existing list of notes of pattern
new_notes = list of empty notes, length new_length
for each note in old_notes:
    replace with a tuple of the note and its index in the list
reverse old_notes
for each index and note pair in old_notes:
```

```
if the note is not an empty note:
    new_index = floor(index * ratio)
    new_notes[new_index] = note
```

The `ratio` variable is the ratio between the old and new length of the pattern. Multiplying the index of a note in the old pattern by the ratio gives the index of that note in the new pattern, as can be seen in the following diagram.

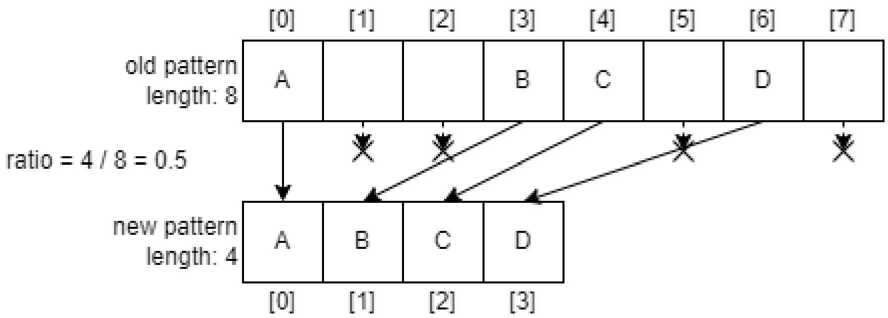


If the multiplication gives a non-integer index, this is truncated, as can be seen in the line `new_index = floor(index * ratio)`. This is not a perfect solution, but is good enough given the restriction of using a tick-based system. This is demonstrated in the next diagram.

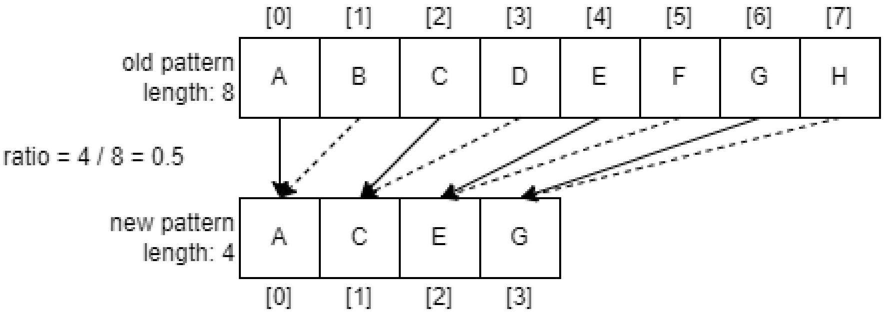


This is fully functional for increasing the length of a pattern. However, when shrinking a pattern from a longer length to a shorter one, multiple list indexes in the old pattern may be mapped to a single list index in the new pattern. The algorithm resolves this in two ways:

- Empty notes are ignored: they are just the absence of a note, so should not overwrite actual notes in the pattern. This is the purpose of the condition `if the note is not an empty note` in the algorithm – if the note is empty then the new pattern is not written to. This is illustrated in the diagram below.



- The list is processed in reverse order. This is primarily down to my personal preference: I wanted a note to overwrite another one if it was earlier in the pattern, as this generally means that on-beat notes overwrite off-beat notes. However, just reversing the list would mean that the original indexes of the notes were lost and would have to be recalculated. To avoid this, I attached the indexes of the notes to the notes themselves in a tuple before reversing the list. In hindsight, there are probably easier and more efficient ways to do this, but this method works fine. This is illustrated in the following diagram.



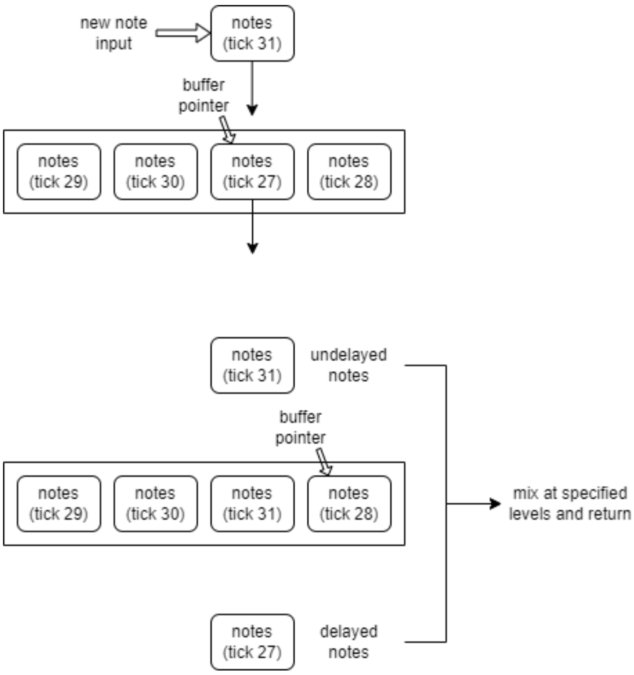
Delay effect

See ***effect_delay.py*** *(p.50)*

The delay effect is the first effect (note processor) that I wrote for the project. Its purpose is to take incoming notes and delay them by a specified number of ticks before outputting them again. If the delayed notes are played alongside the original, unaltered notes, the effect is that of an echo, which may sound pleasing if the volume and pan of the delayed notes are appropriately set.

I used a circular buffer to store the lists of notes that the effect receives on successive ticks. The buffer is the same length as the delay length, and the pointer to the current item in the buffer is incremented every tick, so that when a list of notes is placed into the buffer at the index of the buffer pointer, it will be returned when the buffer pointer next visits that index, which will be the specified number of ticks later.

In this example, the buffer has a length of 4, and the result is that the buffer pointer points to a list of notes that was input 4 ticks ago. Hence, the un-delayed and delayed lists of notes originated 4 ticks apart, as desired.



PROJECT TECHNICAL SOLUTION

Files of interest

See the project design for detailed descriptions of these components.

node.py, events.py, node_actions.py, undo_manager.py, node_editor.py, node_factory.py, model.py Together, these files form the data model of the program. In particular, see **undo_manager.py**, which contains the logic for undo/redo functionality, and **model.py**, which contains the pattern stretching algorithm in the `change_pattern_length()` method.

pattern.py, pattern_group.py, channel.py, channel_group.py, effect.py, song_config.py These are subclasses of the Node class which are used in the hierarchy of song objects. In particular, see **channel.py**, which is where the most complex note processing is performed.

playback.py, tick_manager.py, loop_hijacker.py, audio_generator.py, instrument_sounds.py, audio_player.py Together, these files form the playback logic of the program. In particular, see **loop_hijacker.py** for the GUI library event loop hijacking mechanism, and **audio_generator.py** for the algorithm that turns notes into audio.

effect_delay.py This is where the algorithm for the delay effect is implemented.

easy_effect_ui.py, effect_delay_ui.py *(not described in project design)* The first of these files is a base class which provides methods to easily create user interfaces for effects. The second file shows this base class in action.

Files included

The files comprising my project are listed below in alphabetical order. All classes start with brief descriptions of what they do.

- audio_exporter.py
- audio_generator.py
- audio_player.py
- bar_display.py
- bottom_frame.py
- channel.py
- channel_group.py
- channel_header.py
- channel_header_canvas.py
- easy_effect_ui.py
- effect.py
- effect_add_frame.py
- effect_delay.py
- effect_delay_ui.py
- effect_dummy.py
- effect_dummy_ui.py
- effect_rack.py
- effect_ui.py
- effect_ui_factory.py

- events.py
- instrument_settings.py
- instrument_sounds.py
- loop_hijacker.py
- main.py
- model.py
- node.py
- node_actions.py
- node_editor.py
- node_factory.py
- note.py
- pattern.py
- pattern_group.py
- pattern_list.py
- pattern_settings.py
- piano_notes_canvas.py
- piano_roll.py
- piano_roll_canvas.py
- placement_display.py
- playback.py
- sequencer.py
- song_config.py
- song_settings.py
- tick_manager.py
- top_frame.py
- undo_manager.py

Code listing

I wrote a script to automatically generate the following formatted code listing from my codebase, which is why its appearance on the page is slightly different from the rest of the documentation.

---

audio_exporter.py

```
import numpy as np
import soundfile

from model import Model
from tick_manager import TickManager
from instrument_sounds import InstrumentSounds
from audio_generator import AudioGenerator

class AudioExporter:
    """Turns a song into audio (without waiting for real-time) and writes it to a
.wav file."""

    def __init__(self, model: Model, sounds: InstrumentSounds, block_size: int = 2
400):
        self.model = model
        self.sounds = sounds
```

```
        self.block_size = block_size

    def export(self, filename: str):
        tick_manager = TickManager(model=self.model, ignore_loop=True)
        audio_generator = AudioGenerator(block_size=self.block_size, sounds=self.s
ounds)

        audio_blocks: list[np.ndarray] = []
        tick_manager.set_tick(0, 0)

        # workaround for clearing effects when exporting: just run the buffer a bi
t
        tick_manager.disable_sequence()
        for _ in range(32): # arbitrary number, might need to increase this
            next_tick = tick_manager.next_tick()
            self.model.channel_group.tick(*next_tick)
        tick_manager.enable_sequence()

        while tick_manager.sequence_enabled:
            next_tick = tick_manager.next_tick()
            notes = self.model.channel_group.tick(*next_tick)
            audio_block: np.ndarray = audio_generator.tick(notes)
            audio_blocks.append(audio_block)

        final_audio = np.concatenate(audio_blocks, axis=0)
        soundfile.write(filename, final_audio, samplerate=48000)
```

---

audio_generator.py

```
import numpy as np

from note import Note
from instrument_sounds import InstrumentSounds

# NOTE: this code is reliant on block size being 1/20th of a second
# this is fine and it works but is something to keep in mind

class AudioGenerator():
    """Turns lists of Notes into blocks of audio."""

    def __init__(self, block_size: int, sounds: InstrumentSounds):
        self.block_size = block_size
        self.sounds = sounds
        self.current_sounds: list[tuple[np.ndarray, int]] = []

    def tick(self, notes: list[Note]) -> np.ndarray:
        self.process_new_notes(notes)
        return self.tick_audio()

    def process_new_notes(self, notes: list[Note]):
        for note in notes:
            sound = self.sounds.get_sound(note.instrument, note.pitch) * self.ster
eo_volumes(note)
            self.current_sounds.append((sound, 0))

    def tick_audio(self) -> np.ndarray:
        block = np.zeros((self.block_size, 2), dtype=np.float64)
```

```
        new_current_sounds = []
        for sound, start_index in self.current_sounds:
            end_index = start_index + self.block_size
            block += sound[start_index:end_index]
            if end_index < sound.shape[0]:
                new_current_sounds.append((sound, end_index))
        self.current_sounds = new_current_sounds
        return block

    def stereo_volumes(self, note: Note) -> np.ndarray:
        """Return the volumes of the left and right channels as a numpy array."""
        vol_left = note.volume * max(1 - note.pan, 1)
        vol_right = note.volume * max(1 + note.pan, 1)
        return np.array([vol_left, vol_right], dtype=np.float64)
```

---

audio_player.py

```
import queue
import numpy as np
import sounddevice as sd

class AudioPlayer:
    """Responsible for getting blocks of audio from a queue to a physical output d
evice."""

    def __init__(self, audio_queue: queue.Queue):
        self.audio_queue = audio_queue
        self.stream = None
        self.init_stream(device=sd.default.device[1])

    def init_stream(self, device):
        if isinstance(self.stream, sd.OutputStream):
            self.stream.stop()
        self.stream = sd.OutputStream(
            samplerate=48000,
            blocksize=2400,
            device=device,
            callback=self.sd_callback)
        self.stream.start()

    def sd_callback(self, outdata: np.ndarray, frames: int, time, status):
        try:
            data = self.audio_queue.get_nowait()
        except queue.Empty as e:
            print("no audio data, filling with blanks...")
            outdata.fill(0)
            return
        outdata[:] = data
```

---

bar_display.py

```
import tkinter as tk
import tkinter.ttk as ttk

from node import Node
from events import EventBus, Listener
```

```python
from node_editor import NodeEditor
from model import Model

class BarDisplay(Listener, tk.Canvas):
    """UI component - displays bar numbers and playback start/loop points above th
e sequencer."""

    def __init__(self, parent, *args, model: Model, **kwargs):
        self.model = model
        self.selected_bar: int = 0

        self.strip_height: int = 20
        self.bar_width: int = 60

        self.bg_colour: str = "gray75"
        self.guideline_colour: str = "gray50"
        self.selected_bar_colour: str = "red"

        super().__init__(
            parent,
            *args,
            height=self.strip_height,
            scrollregion=(0, 0, 0, 0),
            highlightthickness=0,
            bg=self.bg_colour,
            **kwargs
        )
        self.bind("<ButtonPress-1>", self.select_bar)
        self.bind("<ButtonPress-3>", self.set_loop_start)
        self.bind("<ButtonRelease-3>", self.set_loop_end)

        self.model.event_bus.add_listener(self)
        self.draw_everything()

    def destroy(self, *args, **kwargs):
        self.model.event_bus.remove_listener(self)
        super().destroy(*args, **kwargs)

    def node_property_set(self, node: Node, key, old_value, new_value):
        if node is self.model.song_config:
            if key == "sequence_length":
                self.draw_everything()
            elif key == "loop_start" or key == "loop_end":
                self.draw_loop_markers()

    def bar_selected(self, bar: int):
        self.selected_bar = bar
        self.draw_start_marker()

    def reset_ui(self):
        self.draw_everything()

    def draw_everything(self):
        self.delete("all")
        self.draw_background()
        self.draw_start_marker()
        self.draw_loop_markers()
```

```python
    def draw_background(self):
        sequence_length = self.model.song_config.get_property("sequence_length")
        self.configure(scrollregion=(0, 0, sequence_length * self.bar_width, 0))
        for bar_number in range(sequence_length):
            self.create_line(
                bar_number * self.bar_width,
                0,
                bar_number * self.bar_width,
                self.strip_height,
                fill=self.guideline_colour,
                width=0
            )
            self.create_text(
                (bar_number * self.bar_width) + 3,
                0,
                text=bar_number,
                anchor="nw",
                fill=self.guideline_colour
            )

    def draw_start_marker(self):
        self.delete("start_marker")
        self.create_polygon(
            (self.selected_bar * self.bar_width) - 5,
            0,
            (self.selected_bar * self.bar_width) + 5,
            0,
            (self.selected_bar * self.bar_width) + 5,
            self.strip_height - 5,
            (self.selected_bar * self.bar_width),
            self.strip_height,
            (self.selected_bar * self.bar_width) - 5,
            self.strip_height - 5,
            fill=self.selected_bar_colour,
            tags="start_marker"
        )

    def draw_loop_markers(self):
        self.delete("loop_marker")
        self.create_rectangle(
            self.model.song_config.get_property("loop_start") * self.bar_width,
            0,
            self.model.song_config.get_property("loop_end") * self.bar_width,
            self.strip_height,
            fill="purple",
            stipple="gray50",
            tags="loop_marker"
        )
        self.tag_raise("start_marker", "loop_marker")

    def select_bar(self, event: tk.Event):
        bar = self.get_bar_at_coords(event.x)
        self.model.event_bus.bar_selected(bar)

    def set_loop_start(self, event: tk.Event):
        bar = self.get_bar_at_coords(event.x)
        self.model.ed.set_property(self.model.song_config, "loop_start", bar)

    def set_loop_end(self, event: tk.Event):
```

```
        bar = self.get_bar_at_coords(event.x)
        self.model.ed.set_property(self.model.song_config, "loop_end", bar)

    def get_bar_at_coords(self, x: int) -> int:
        canvas_x = self.canvasx(x)
        bar = int(canvas_x // self.bar_width)
        return bar
```

---

bottom_frame.py

```
import tkinter as tk
import tkinter.ttk as ttk

from model import Model

from piano_roll import PianoRoll
from effect_rack import EffectRack

class BottomFrame(ttk.Frame):
    """UI component - contains the piano roll and effect rack, with tabs to switch
between them."""

    def __init__(self, parent, *args, model: Model, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.model = model
        self.columnconfigure(1, weight=1)
        self.showing: bool = True

        self.btn_hide = ttk.Button(
            self,
            text="▼",
            width=3,
            command=self.toggle_show
        )

        self.notebook = ttk.Notebook(self)

        self.piano_roll = PianoRoll(self.notebook, model=self.model)
        self.notebook.add(self.piano_roll, text="piano roll")

        self.effect_rack = EffectRack(self.notebook, model=self.model)
        self.notebook.add(self.effect_rack, text="effects")

        self.btn_hide.grid(column=0, row=0, sticky="n")
        self.notebook.grid(column=1, row=0, sticky="nsew")

    def toggle_show(self):
        if self.showing:
            self.notebook.grid_forget()
            self.btn_hide.configure(text="▲")
            self.showing = False
        else:
            self.notebook.grid(column=1, row=0, sticky="nsew")
            self.btn_hide.configure(text="▼")
            self.showing = True
```

---

channel.py

```
from node import Node
from note import Note
from pattern import Pattern
from pattern_group import PatternGroup
from effect import Effect

class Channel(Node):
    """Song object - analogous to a voice in a song."""

    def __init__(self, *args, pattern_group: PatternGroup, sequence_length: int =
20, **kwargs):
        super().__init__(*args, **kwargs)
        self._set_property("name", "channel name")
        self._set_property("colour", "gray50")
        self._set_property("main_instrument", 0)
        self._set_property("sustain_enabled", False)
        self._set_property("sustain_instrument", 0)
        self._set_property("sustain_mix", 0.5)
        self._set_property("volume", 1.0)
        self._set_property("pan", 0.0)
        self._set_property("mute", False)
        self._set_property("solo", False)
        self._set_property("placements", [-1] * sequence_length)
        self.pattern_group = pattern_group
        self.sustained_note = None

    def tick(self, mono_tick: int, sequence_enabled: bool, bar_number: int, pat_ti
ck: int) -> list[Note]:
        if not sequence_enabled: # NO-PATTERN TICK!
            note_numbers = [-2]
        else:
            # get note numbers from pattern
            pattern_id = self.get_property("placements")[bar_number]
            if pattern_id == -1: # no pattern
                note_numbers = [-1]
            else:
                pattern: Pattern = self.pattern_group.get_child_by_id(pattern_id)
                note_numbers = pattern.get_notes(pat_tick)

        # convert to Note objects
        notes: list[Note] = self.convert_numbers_to_notes(note_numbers)

        # apply effects in sequence
        for effect in self.children_iterator():
            if isinstance(effect, Effect):
                notes = effect.tick(notes, mono_tick)

        # apply volume and pan
        notes = [note.apply_volume_and_pan(
            volume=self.get_property("volume"),
            pan=self.get_property("pan")
        ) for note in notes]

        return notes


    def convert_numbers_to_notes(self, note_numbers: list[int]) -> list[Note]:
```

```
        notes = []
        sustain_enabled = self.get_property("sustain_enabled")
        for note_number in note_numbers:
            if 0 <= note_number <= 24: # normal note
                notes.append(Note(
                    instrument=self.get_property("main_instrument"),
                    pitch=note_number
                ))
                self.sustained_note = note_number
            elif note_number == -1: # no note
                if sustain_enabled and self.sustained_note is not None:
                    notes.append(Note(
                        instrument=self.get_property("sustain_instrument"),
                        pitch=self.sustained_note,
                        volume=self.get_property("sustain_mix")
                    ))
            elif note_number == -2: # SUSTAIN OFF!
                self.sustained_note = None
        return notes
```

---

channel_group.py

```
from node import Node
from note import Note
from channel import Channel

class ChannelGroup(Node):
    """Song object - holds a song's channels, and deals with mute/solo functionali
ty."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def tick(self, mono_tick: int, sequence_enabled: bool, bar_number: int, pat_ti
ck: int) -> list[Note]:
        notes: list[Note] = []
        solo_active = any(child.get_property("solo") for child in self.children_it
erator())
        for child in self.children_iterator():
            if isinstance(child, Channel):
                channel_notes = child.tick(mono_tick, sequence_enabled, bar_number
, pat_tick)
                muted = child.get_property("mute")
                soloed = child.get_property("solo")
                if soloed or (not muted and not solo_active):
                    notes.extend(channel_notes)
        return notes
```

---

channel_header.py

```
import tkinter as tk
import tkinter.ttk as ttk
from tkinter import colorchooser

from node import Node
from events import EventBus, Listener
```

```
from node_editor import NodeEditor
from model import Model

from channel import Channel

class ChannelHeader(Listener, ttk.Frame):
    """UI component - gives access to channel settings next to the sequencer."""

    def __init__(self, parent, *args, model: Model, channel: Channel, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.model = model
        self.channel = channel

        self.init_ui()
        self.model.event_bus.add_listener(self)
        self.bind("<ButtonPress-1>", lambda e: self.model.event_bus.node_selected(
self.channel))
        self.update_ui()

    def destroy(self, *args, **kwargs):
        self.model.event_bus.remove_listener(self)
        super().destroy(*args, **kwargs)

    def node_property_set(self, node: Node, key, old_value, new_value):
        if node is self.channel:
            self.update_ui()

    padding = {"padx": 2, "pady": 2}

    def choose_colour(self, event: tk.Event):
        colour = colorchooser.askcolor()[1]
        if colour is not None: self.model.ed.set_property(self.channel, "colour",
colour)

    def init_ui(self):
        self.top_frame = ttk.Frame(self)

        self.lbl_colour = ttk.Label(self.top_frame, width=3)
        self.lbl_colour.bind(
            "<ButtonPress-1>",
            self.choose_colour
        )

        self.var_name = tk.StringVar(self)
        self.inp_name = ttk.Entry(
            self.top_frame,
            width=15,
            textvariable=self.var_name,
        )
        self.inp_name.bind(
            "<Return>",
            lambda e: self.model.ed.set_property(self.channel, "name", self.var_na
me.get())
        )

        self.btn_delete = ttk.Button(
            self.top_frame,
            text="🗑",
```

```
            width=3,
            command=lambda: self.model.ed.remove_child(self.model.channel_group, s
elf.channel)
        )

        self.bottom_frame = ttk.Frame(self)

        self.lbl_volume = ttk.Label(self.bottom_frame, text="V:")
        self.var_volume = tk.DoubleVar(self, value=1.0)
        self.inp_volume = ttk.Spinbox(
            self.bottom_frame,
            from_=0.0, to=1.0, increment=0.1,
            width=5,
            textvariable=self.var_volume,
            command=lambda: self.model.ed.set_property(self.channel, "volume", sel
f.var_volume.get())
        )
        self.inp_volume.bind(
            "<Return>",
            lambda e: self.model.ed.set_property(self.channel, "volume", self.var_
volume.get())
        )

        self.lbl_pan = ttk.Label(self.bottom_frame, text="P:")
        self.var_pan = tk.DoubleVar(self, value=0.0)
        self.inp_pan = ttk.Spinbox(
            self.bottom_frame,
            from_=-1.0, to=1.0, increment=0.1,
            width=5,
            textvariable=self.var_pan,
            command=lambda: self.model.ed.set_property(self.channel, "pan", self.v
ar_pan.get())
        )
        self.inp_pan.bind(
            "<Return>",
            lambda e: self.model.ed.set_property(self.channel, "pan", self.var_pan
.get())
        )

        self.btn_mute = ttk.Button(
            self,
            text="M",
            width=3,
            command=lambda: self.model.ed.toggle_bool(self.channel, "mute")
        )
        self.btn_solo = ttk.Button(
            self,
            text="S",
            width=3,
            command=lambda: self.model.ed.toggle_bool(self.channel, "solo")
        )

        self.btn_move_up = ttk.Button(
            self,
            text="▲",
            width=2,
            command=lambda: self.move_channel(-1)
        )
        self.btn_move_down = ttk.Button(
```

```
            self,
            text="▼",
            width=2,
            command=lambda: self.move_channel(1)
        )

        self.lbl_colour.grid(column=0, row=0, **self.padding)
        self.inp_name.grid(column=1, row=0, **self.padding)
        self.btn_delete.grid(column=2, row=0, **self.padding)
        self.top_frame.grid(column=0, row=0, sticky="w")

        self.lbl_volume.grid(column=0, row=0, **self.padding)
        self.inp_volume.grid(column=1, row=0, **self.padding)
        self.lbl_pan.grid(column=2, row=0, **self.padding)
        self.inp_pan.grid(column=3, row=0, **self.padding)
        self.bottom_frame.grid(column=0, row=1, sticky="w")

        self.btn_mute.grid(column=1, row=0, **self.padding)
        self.btn_solo.grid(column=1, row=1, **self.padding)
        self.btn_move_up.grid(column=2, row=0, **self.padding)
        self.btn_move_down.grid(column=2, row=1, **self.padding)

    def update_ui(self):
        self.lbl_colour.config(background=self.channel.get_property("colour"))
        self.btn_mute.state(["pressed" if self.channel.get_property("mute") else "
!pressed"])
        self.btn_solo.state(["pressed" if self.channel.get_property("solo") else "
!pressed"])
        self.var_name.set(self.channel.get_property("name"))
        self.var_volume.set(self.channel.get_property("volume"))
        self.var_pan.set(self.channel.get_property("pan"))

    def move_channel(self, delta: int):
        old_index = self.model.channel_group.get_index_of_child(self.channel)
        new_index = old_index + delta
        self.model.ed.move_child(self.model.channel_group, old_index, new_index)
```

---

channel_header_canvas.py

```
import tkinter as tk
import tkinter.ttk as ttk

from node import Node
from events import Listener
from model import Model

from channel_header import ChannelHeader

class ChannelHeaderCanvas(Listener, tk.Canvas):
    """UI component - provides a scrollable canvas for channel headers."""

    def __init__(self, parent, *args, model: Model, **kwargs):
        self.model = model
        self.bg_colour: str = "gray75"
        self.header_height: int = 60

        super().__init__(
```

```python
            parent,
            *args,
            scrollregion=(0, 0, 0, 0),
            highlightthickness=0,
            bg=self.bg_colour,
            **kwargs
        )

        self.internal_frame = ttk.Frame(self)
        self.create_window(0, 0, anchor="nw", window=self.internal_frame)
        self.internal_frame.bind(
            "<Configure>",
            lambda e: self.configure(width=self.internal_frame.winfo_reqwidth())
        )

        self.model.event_bus.add_listener(self)
        self.update_ui()

    def destroy(self, *args, **kwargs):
        self.model.event_bus.remove_listener(self)
        super().destroy(*args, **kwargs)

    def node_child_added(self, parent: Node, child: Node, id: int, index: int):
        if parent is self.model.channel_group:
            self.update_ui()

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        if parent is self.model.channel_group:
            self.update_ui()

    def node_child_moved(self, parent: Node, old_index: int, new_index: int):
        if parent is self.model.channel_group:
            self.update_ui()

    def reset_ui(self):
        self.update_ui()

    def update_ui(self):
        for header in self.internal_frame.winfo_children():
            header.destroy()

        channel_count = self.model.channel_group.children_count()
        self.configure(scrollregion=(0, 0, 0, self.header_height * channel_count))

        for index, channel in enumerate(self.model.channel_group.children_iterator
()):
            header = ChannelHeader(self.internal_frame, model=self.model, channel=
channel)
            header.grid(column=0, row=index, sticky="nsew")
            self.internal_frame.rowconfigure(index, minsize=self.header_height)
```

---

easy_effect_ui.py

```python
from abc import abstractmethod
from typing import Callable
import tkinter as tk
import tkinter.ttk as ttk
```

```python
from effect_ui import EffectUI

class EasyEffectUI(EffectUI):
    """UI component - base class providing methods to easily create simple effect
UIs."""

    effect_name: str = "easy effect ui"
    ui_width: int = 200

    instrument_names = [
        "harp",
        "basedrum",
        "snare",
        "hat",
        "bass",
        "flute",
        "bell",
        "guitar",
        "chime",
        "xylophone",
        "iron_xylophone",
        "cow_bell",
        "didgeridoo",
        "bit",
        "banjo",
        "pling"
    ]
    grid_kwargs = {"sticky": "w", "padx": 2, "pady": 2}

    @abstractmethod
    def init_ui(self):
        super().init_ui()
        self.next_grid_row: int = 1
        self.update_callbacks: list[Callable] = []

    @abstractmethod
    def update_ui(self):
        super().update_ui()
        for callback in self.update_callbacks:
            callback()

    def add_label(self, text: str):
        label = ttk.Label(self, text=text)
        label.grid(column=0, row=self.next_grid_row, **self.grid_kwargs)
        self.next_grid_row += 1

    def add_checkbox(self, name: str, key: str):
        var = tk.BooleanVar()
        set_callback = lambda e=None: self.model.ed.set_property(self.effect, key,
var.get())
        get_callback = lambda: var.set(self.effect.get_property(key))
        checkbox = ttk.Checkbutton(self, text=name, command=set_callback, variable
=var)
        checkbox.grid(column=0, row=self.next_grid_row, **self.grid_kwargs)
        self.next_grid_row += 1
        self.update_callbacks.append(get_callback)
```

```
    def add_spinbox(self, name: str, key: str, low: float, high: float, step: floa
t, int_only: bool = False):
        var = (tk.IntVar() if int_only else tk.DoubleVar())
        set_callback = lambda e=None: self.model.ed.set_property(
            self.effect, key, self._clamp(var.get(), low, high))
        get_callback = lambda: var.set(self.effect.get_property(key))
        frame = ttk.Frame(self)
        label = ttk.Label(frame, text=name)
        spinbox = ttk.Spinbox(
            frame, from_=low, to=high, increment=step, width=5,
            textvariable=var,
            command=set_callback)
        spinbox.bind("<Return>", set_callback)
        label.grid(column=0, row=0)
        spinbox.grid(column=1, row=0)
        frame.grid(column=0, row=self.next_grid_row, **self.grid_kwargs)
        self.next_grid_row += 1
        self.update_callbacks.append(get_callback)

    def add_instrument_choice(self, name: str, key: str):
        var = tk.StringVar()
        set_callback = lambda e=None: self.model.ed.set_property(self.effect, key,
            self.instrument_names.index(var.get()))
        get_callback = lambda: var.set(self.instrument_names[self.effect.get_prope
rty(key)])
        frame = ttk.Frame(self)
        label = ttk.Label(frame, text=name)
        combo = ttk.OptionMenu(
            frame, var,
            self.instrument_names[0],
            *self.instrument_names,
            command=set_callback)
        label.grid(column=0, row=0)
        combo.grid(column=1, row=0)
        frame.grid(column=0, row=self.next_grid_row, **self.grid_kwargs)
        self.next_grid_row += 1
        self.update_callbacks.append(get_callback)

    def _clamp(self, value, low, high):
        return min(max(value, low), high)
```

---

effect.py

```
from abc import ABC, abstractmethod
from node import Node
from note import Note

class Effect(Node):
    """Song object - base class for all effects."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._set_property("enabled", True)

    def tick(self, notes: list[Note], mono_tick: int) -> list[Note]:
        # NOTE: this is for internal use with the "enabled" property
        if self.get_property("enabled"):
```

```
            return self.process_notes(notes, mono_tick)
        else:
            return notes

    @abstractmethod
    def process_notes(self, notes: list[Note], mono_tick: int) -> list[Note]:
        ...
```

---

effect_add_frame.py

```
import tkinter as tk
import tkinter.ttk as ttk

from node import Node
from events import Listener
from model import Model

from channel import Channel

from effect_dummy import EffectDummy
from effect_delay import EffectDelay

class EffectAddFrame(Listener, ttk.Frame):
    """UI component - used to add effects to a channel's effect rack."""

    effects = {
        "dummy": EffectDummy,
        "delay": EffectDelay,
    }
    effect_names = list(effects.keys())

    def __init__(self, parent, model: Model, **kwargs):
        super().__init__(parent, **kwargs)
        self.model = model
        self.channel: Channel | None = None

        self.init_ui()
        self.model.event_bus.add_listener(self)
        self.update_ui()

    def destroy(self, *args, **kwargs):
        self.model.event_bus.remove_listener(self)
        super().destroy(*args, **kwargs)

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        if child is self.channel:
            self.channel = None
            self.update_ui()

    def node_selected(self, node: Node):
        if isinstance(node, Channel):
            self.channel = node
            self.update_ui()

    def reset_ui(self):
        self.channel = None
        self.update_ui()
```

```python
    def init_ui(self):
        self.var_effect = tk.StringVar(self)
        self.cmb_effect = ttk.OptionMenu(
            self,
            self.var_effect,
            self.effect_names[0],
            *self.effect_names
        )
        self.btn_add_effect = ttk.Button(
            self,
            text="+ add effect",
            command=lambda: self.add_effect(self.var_effect.get())
        )

        self.cmb_effect.grid(column=0, row=0, sticky="ew")
        self.btn_add_effect.grid(column=0, row=1, sticky="ew")

    def add_effect(self, name: str):
        effect_class = self.effects[name]
        effect_node = effect_class()
        self.model.ed.add_child(self.channel, effect_node)

    def update_ui(self):
        if self.channel is None:
            self.set_all_states(["disabled"])
        else:
            self.set_all_states(["!disabled"])

    def set_all_states(self, statespec: list[str]):
        for component in (self.cmb_effect, self.btn_add_effect):
            component.state(statespec)
```

---

effect_delay.py

```python
from note import Note
from effect import Effect

class EffectDelay(Effect):
    """Song object - delays notes, creating an echo-like sound."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._set_property("delay_ticks", 4)
        self._set_property("dry_mix", 1.0)
        self._set_property("wet_mix", 0.5)
        self._set_property("wet_pan", 0.0)
        self.notes_buffer: list[list[Note]] = [[]]
        self.buffer_index: int = 0

    def process_notes(self, notes: list[Note], mono_tick: int) -> list[Note]:
        # the ol' buffer switcheroo
        delayed_notes = self.notes_buffer[self.buffer_index]
        self.notes_buffer[self.buffer_index] = notes

        # mix things
        dry_notes = [note.apply_volume_and_pan(
```

```
            volume=self.get_property("dry_mix")
        ) for note in notes]
        wet_notes = [note.apply_volume_and_pan(
            volume=self.get_property("wet_mix"),
            pan=self.get_property("wet_pan")
        ) for note in delayed_notes]

        # advance buffer index
        self.buffer_index += 1
        if self.buffer_index >= self.get_property("delay_ticks"): # loop back roun
d
            self.buffer_index = 0
        elif self.buffer_index >= len(self.notes_buffer): # extend buffer
            self.notes_buffer.append([])

        # output
        return dry_notes + wet_notes
```

---

effect_delay_ui.py

```
from easy_effect_ui import EasyEffectUI

class EffectDelayUI(EasyEffectUI):
    """UI component - controls for the delay effect."""

    effect_name: str = "delay"
    ui_width: int = 150

    def init_ui(self):
        super().init_ui()
        self.add_spinbox("delay (ticks):", "delay_ticks", 1, 32, 1, int_only=True)
        self.add_spinbox("dry mix:", "dry_mix", 0.0, 1.0, 0.1)
        self.add_spinbox("wet mix:", "wet_mix", 0.0, 1.0, 0.1)
        self.add_spinbox("wet pan:", "wet_pan", -1.0, 1.0, 0.1)

    def update_ui(self):
        super().update_ui()
```

---

effect_dummy.py

```
from note import Note
from effect import Effect

class EffectDummy(Effect):
    """Song object - demo effect, does nothing."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # initialize properties and state here

    def process_notes(self, notes: list[Note], mono_tick: int) -> list[Note]:
        # do tick logic here
        return notes
```

---

effect_dummy_ui.py
```
import tkinter as tk
import tkinter.ttk as ttk

from effect_ui import EffectUI

class EffectDummyUI(EffectUI):
    """UI component - UI for the dummy effect."""

    effect_name = "dummy effect"
    ui_width = 200

    def init_ui(self):
        super().init_ui()
        # initialize UI components here - grid into column 0, row 1
        self.label = ttk.Label(self, text="dummy effect")
        self.label.grid(column=0, row=1, sticky="nsew", padx=5, pady=5)

    def update_ui(self):
        super().update_ui()
        ... # update UI components based on self.effect (which will not be None)
```

---

effect_rack.py
```
import tkinter as tk
import tkinter.ttk as ttk

from node import Node
from events import Listener
from model import Model
from channel import Channel
from effect import Effect
from effect_ui_factory import EffectUIFactory
from instrument_settings import InstrumentSettings
from effect_add_frame import EffectAddFrame

class EffectRack(Listener, ttk.Frame):
    """UI component - displays a channel's effects."""

    def __init__(self, parent, model: Model, **kwargs):
        super().__init__(parent, **kwargs)
        self.model = model
        self.channel: Channel | None = None

        self.init_ui()
        self.factory = EffectUIFactory(parent=self.internal_frame, model=self.mode
l)

        self.model.event_bus.add_listener(self)
        self.update_ui()

    def destroy(self):
        self.model.event_bus.remove_listener(self)
        super().destroy()

    def node_child_added(self, parent: Node, child: Node, id: int, index: int):
```

```
        if parent is self.channel:
            self.update_ui()

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        if parent is self.channel:
            self.update_ui()
        elif child is self.channel:
            self.channel = None
            self.update_ui()

    def node_child_moved(self, parent: Node, old_index: int, new_index: int):
        if parent is self.channel:
            self.update_ui()

    def node_selected(self, node: Node):
        if isinstance(node, Channel):
            self.channel = node
            self.update_ui()

    def reset_ui(self):
        self.channel = None
        self.update_ui()

    def init_ui(self):
        self.columnconfigure(1, weight=1)
        self.rowconfigure(0, weight=1)

        self.instrument_settings = InstrumentSettings(self, model=self.model)
        self.instrument_settings.grid(column=0, row=0, sticky="nsew", padx=5, pady
=5)

        self.effect_add_frame = EffectAddFrame(self, model=self.model)
        self.effect_add_frame.grid(column=2, row=0, sticky="nsew", padx=5, pady=5)

        self.canvas = tk.Canvas(self)
        self.canvas.grid(column=1, row=0, sticky="nsew")

        self.internal_frame = ttk.Frame(self.canvas)
        self.canvas.create_window(0, 0, anchor="nw", window=self.internal_frame)

        self.internal_frame.bind(
            "<Configure>",
            lambda e: self.canvas.configure(scrollregion=(0, 0, self.internal_fram
e.winfo_reqwidth(), 0))
        )

        self.scrollbar = ttk.Scrollbar(self, orient=tk.HORIZONTAL)
        self.canvas.configure(xscrollcommand=self.scrollbar.set)
        self.scrollbar["command"] = self.canvas.xview
        self.scrollbar.grid(column=1, row=1, sticky="ew")

    def update_ui(self):
        for child in self.internal_frame.winfo_children():
            child.destroy()

        if self.channel is not None:
            for index, effect in enumerate(self.channel.children_iterator()):
                if isinstance(effect, Effect):
```

```
                    effect_ui = self.factory.create_ui(effect)
                    effect_ui.grid(column=index, row=0, sticky="ns", padx=5, pady=
5)
```

---

effect_ui.py

```
from abc import ABC, abstractmethod
import tkinter as tk
import tkinter.ttk as ttk

from node import Node
from events import Listener
from model import Model
from effect import Effect

class EffectUI(Listener, ttk.Labelframe, ABC):
    """UI component - base class of effect UIs, supplies effect on/off/reorder/del
ete controls."""

    effect_name: str = "base effect ui"
    ui_width: int = 200

    def __init__(self, parent, model: Model, effect: Effect, **kwargs):
        super().__init__(parent, text=self.effect_name, **kwargs)
        self.model = model
        self.effect = effect

        self.init_ui()
        self.model.event_bus.add_listener(self)
        self.update_ui()

    def destroy(self):
        self.model.event_bus.remove_listener(self)
        super().destroy()

    def node_property_set(self, node: Node, key, old_value, new_value):
        if node is self.effect:
            self.update_ui()

    @abstractmethod
    def init_ui(self):
        pad_kwargs = {"padx": 2, "pady": 2}
        self.columnconfigure(0, minsize=self.ui_width)
        self.header_frame = ttk.Frame(self)

        self.btn_enabled = ttk.Button(
            self.header_frame,
            text="⬤",
            width=3,
            command=lambda: self.model.ed.toggle_bool(self.effect, "enabled")
        )

        self.btn_move_left = ttk.Button(
            self.header_frame,
            text="◀",
            width=3,
            command=lambda: self.move_effect(-1)
```

```python
        )
        self.btn_move_right = ttk.Button(
            self.header_frame,
            text="▶",
            width=3,
            command=lambda: self.move_effect(1)
        )
        self.btn_delete = ttk.Button(
            self.header_frame,
            text="🗑",
            width=3,
            command=lambda: self.model.ed.remove_child(self.effect.parent, self.ef
fect)
        )

        self.header_frame.columnconfigure(1, weight=1)
        self.btn_enabled.grid(column=0, row=0, **pad_kwargs)
        self.btn_move_left.grid(column=2, row=0, **pad_kwargs)
        self.btn_move_right.grid(column=3, row=0, **pad_kwargs)
        self.btn_delete.grid(column=4, row=0, **pad_kwargs)
        self.header_frame.grid(column=0, row=0, sticky="ew")

    @abstractmethod
    def update_ui(self):
        self.btn_enabled.configure(text=("🔵" if self.effect.get_property("enable
d") else "◯"))

    def move_effect(self, delta: int):
        channel = self.effect.parent
        if channel is not None:
            old_index = channel.get_index_of_child(self.effect)
            new_index = old_index + delta
            self.model.ed.move_child(channel, old_index, new_index)
```

effect_ui_factory.py

```python
from typing import Type

from model import Model
from effect import Effect
from effect_ui import EffectUI

from effect_dummy_ui import EffectDummyUI
from effect_delay_ui import EffectDelayUI
# TODO: import EffectUI subclasses here

class EffectUIFactory:
    """Takes Effects and creates their corresponding UI components."""

    ui_classes: dict[str, Type[EffectUI]] = {
        "EffectDummy": EffectDummyUI,
        "EffectDelay": EffectDelayUI
        # TODO: add entries for EffectUI subclasses here
    }

    def __init__(self, parent, model: Model):
        self.parent = parent
```

```python
        self.model = model

    def create_ui(self, effect: Effect, **kwargs):
        ui_class = self.ui_classes.get(effect.__class__.__name__, None)
        if ui_class is None: return None
        ui_object = ui_class(self.parent, model=self.model, effect=effect, **kwarg
s)
        return ui_object
```

---

events.py

```python
from abc import ABC
from node import Node

class Listener(ABC):
    """Base class for listening to events."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def node_property_set(self, node: Node, key, old_value, new_value):
        ...

    def node_child_added(self, parent: Node, child: Node, id: int, index: int):
        ...

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        ...

    def node_child_moved(self, parent: Node, old_index: int, new_index: int):
        ...

    def node_selected(self, node: Node):
        ...

    def bar_selected(self, bar: int):
        ...

    def bar_playing(self, bar: int):
        ...

    def reset_ui(self):
        # Used when a project is loaded from file
        # All links to old Nodes need to be broken
        ...

class EventBus:
    """
    Facilitates indirect communication between objects,
    based only on what information they need from each other.
    """

    def __init__(self):
        self.listeners: list[Listener] = []

    def add_listener(self, listener: Listener):
        if listener not in self.listeners:
```

```python
        self.listeners.append(listener)

    def remove_listener(self, listener: Listener):
        if listener in self.listeners:
            self.listeners.remove(listener)

    def clear_listeners(self):
        self.listeners = []

    def node_property_set(self, node: Node, key, old_value, new_value):
        for listener in self.listeners:
            listener.node_property_set(node, key, old_value, new_value)

    def node_child_added(self, parent: Node, child: Node, id: int, index: int):
        for listener in self.listeners:
            listener.node_child_added(parent, child, id, index)

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        for listener in self.listeners:
            listener.node_child_removed(parent, child, id, index)

    def node_child_moved(self, parent: Node, old_index: int, new_index: int):
        for listener in self.listeners:
            listener.node_child_moved(parent, old_index, new_index)

    def node_selected(self, node: Node):
        for listener in self.listeners:
            listener.node_selected(node)

    def bar_selected(self, bar: int):
        for listener in self.listeners:
            listener.bar_selected(bar)

    def bar_playing(self, bar: int):
        for listener in self.listeners:
            listener.bar_playing(bar)

    def reset_ui(self):
        for listener in self.listeners:
            listener.reset_ui()
```

---

instrument_settings.py

```python
import tkinter as tk
import tkinter.ttk as ttk

from node import Node
from events import Listener
from model import Model

from channel import Channel

class InstrumentSettings(Listener, ttk.Frame):
    """UI component - controls for a channel's main and sustain instruments."""

    instrument_names = [
        "harp",
```

```
        "basedrum",
        "snare",
        "hat",
        "bass",
        "flute",
        "bell",
        "guitar",
        "chime",
        "xylophone",
        "iron_xylophone",
        "cow_bell",
        "didgeridoo",
        "bit",
        "banjo",
        "pling",
    ]

    padding = {"padx": 5, "pady": 5}

    def __init__(self, parent, model: Model, **kwargs):
        super().__init__(parent, **kwargs)
        self.model = model
        self.channel: Channel | None = None

        self.columnconfigure(0, weight=1)

        self.init_ui()
        self.model.event_bus.add_listener(self)
        self.update_ui()

    def destroy(self, *args, **kwargs):
        self.model.event_bus.remove_listener(self)
        super().destroy(*args, **kwargs)

    def node_property_set(self, node: Node, key, old_value, new_value):
        if node is self.channel:
            self.update_ui()

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        if child is self.channel:
            self.channel = None
            self.update_ui()

    def node_selected(self, node: Node):
        if isinstance(node, Channel):
            self.channel = node
            self.update_ui()

    def reset_ui(self):
        self.channel = None
        self.update_ui()

    def init_ui(self):
        self.var_name = tk.StringVar(self)
        self.inp_name = ttk.Entry(
            self,
            width=20,
            textvariable=self.var_name,
```

```
        )
        self.inp_name.bind(
            "<Return>",
            lambda e: self.model.ed.set_property(self.channel, "name", self.var_na
me.get())
        )

        self.lf_main = ttk.Labelframe(self, text="main instrument")

        self.var_main_instrument = tk.StringVar(self.lf_main)
        self.cmb_main_instrument = ttk.OptionMenu(
            self.lf_main,
            self.var_main_instrument,
            self.instrument_names[0],
            *self.instrument_names,
            command=lambda e: self.model.ed.set_property(
                self.channel,
                "main_instrument",
                self.instrument_names.index(self.var_main_instrument.get()))
        )

        self.lf_sustain = ttk.Labelframe(self, text="sustain instrument")

        self.var_sustain_enabled = tk.BooleanVar(self.lf_sustain)
        self.chk_sustain_enabled = ttk.Checkbutton(
            self.lf_sustain,
            text="enabled?",
            variable=self.var_sustain_enabled,
            command=lambda: self.model.ed.set_property(
                self.channel,
                "sustain_enabled",
                self.var_sustain_enabled.get()
            )
        )

        self.var_sustain_instrument = tk.StringVar(self.lf_sustain)
        self.cmb_sustain_instrument = ttk.OptionMenu(
            self.lf_sustain,
            self.var_sustain_instrument,
            self.instrument_names[0],
            *self.instrument_names,
            command=lambda e: self.model.ed.set_property(
                self.channel,
                "sustain_instrument",
                self.instrument_names.index(self.var_sustain_instrument.get()))
        )

        self.lbl_sustain_mix = ttk.Label(self.lf_sustain, text="mix:")
        self.var_sustain_mix = tk.DoubleVar(self.lf_sustain)
        self.inp_sustain_mix = ttk.Spinbox(
            self.lf_sustain,
            from_=0.0, to=1.0, increment=0.1,
            width=5,
            textvariable=self.var_sustain_mix,
            command=lambda: self.model.ed.set_property(self.channel, "sustain_mix"
, self.var_sustain_mix.get())
        )
        self.inp_sustain_mix.bind(
            "<Return>",
```

```
            lambda e: self.model.ed.set_property(self.channel, "sustain_mix", self
.var_sustain_mix.get())
        )

        self.cmb_main_instrument.grid(column=0, row=0, sticky="w", **self.padding)
        self.chk_sustain_enabled.grid(column=0, row=0, columnspan=2, **self.paddin
g)
        self.cmb_sustain_instrument.grid(column=0, row=1, columnspan=2, sticky="w"
, **self.padding)
        self.lbl_sustain_mix.grid(column=0, row=2, **self.padding)
        self.inp_sustain_mix.grid(column=1, row=2, **self.padding)
        self.inp_name.grid(column=0, row=0, sticky="ew", **self.padding)
        self.lf_main.grid(column=0, row=1, sticky="ew", **self.padding)
        self.lf_sustain.grid(column=0, row=2, sticky="ew", **self.padding)

    def update_ui(self):
        if self.channel is None:
            self.set_all_states(["disabled"])
        else:
            self.set_all_states(["!disabled"])
            self.var_name.set(self.channel.get_property("name"))
            self.var_main_instrument.set(self.instrument_names[self.channel.get_pr
operty("main_instrument")])
            self.var_sustain_enabled.set(self.channel.get_property("sustain_enable
d"))
            self.var_sustain_instrument.set(self.instrument_names[self.channel.get
_property("sustain_instrument")])
            self.var_sustain_mix.set(self.channel.get_property("sustain_mix"))
            for component in (self.cmb_sustain_instrument, self.inp_sustain_mix, s
elf.lbl_sustain_mix):
                component.state(["!disabled" if self.channel.get_property("sustain
_enabled") else "disabled"])

    def set_all_states(self, statespec: list[str]):
        for component in (self.inp_name, self.cmb_main_instrument, self.chk_sustai
n_enabled,
                self.cmb_sustain_instrument, self.lbl_sustain_mix, self.inp_sustai
n_mix):
            component.state(statespec)
```

---

instrument_sounds.py

```
import math
import numpy as np
import samplerate
import soundfile

instrument_paths = {
    0: "sounds/wav/harp.wav",
    1: "sounds/wav/basedrum.wav",
    2: "sounds/wav/snare.wav",
    3: "sounds/wav/hat.wav",
    4: "sounds/wav/bass.wav",
    5: "sounds/wav/flute.wav",
    6: "sounds/wav/bell.wav",
    7: "sounds/wav/guitar.wav",
    8: "sounds/wav/chime.wav",
    9: "sounds/wav/xylophone.wav",
```

```
    10: "sounds/wav/iron_xylophone.wav",
    11: "sounds/wav/cow_bell.wav",
    12: "sounds/wav/didgeridoo.wav",
    13: "sounds/wav/bit.wav",
    14: "sounds/wav/banjo.wav",
    15: "sounds/wav/pling.wav",
}

class InstrumentSounds:
    '''Loads every instrument at every pitch into memory. Does not actually consum
e much memory.'''

    def __init__(self, block_size: int):
        self.block_size = block_size
        self.pitch_ratios = tuple(2 * math.pow(2, -pitch/12) for pitch in range(25
))
        self._sounds: dict[int, list[np.ndarray]] = {}
        for id, path in instrument_paths.items():
            print(f"loading instrument id {id}...")
            self._sounds[id] = []
            sound, _ = soundfile.read(path, dtype="float64", always_2d=True) # aut
omatically scales to [-1, 1]
            for ratio in self.pitch_ratios:
                pitched_sound = samplerate.resample(sound, ratio, "sinc_best")
                pad_length = self.block_size - (pitched_sound.shape[0] % self.bloc
k_size)
                padded_sound = np.pad(pitched_sound, pad_width=((0,pad_length),(0,
0)))
                self._sounds[id].append(padded_sound)
                # print(f"Loaded instrument id {id} with ratio {ratio}")

    def get_sound(self, instrument: int, pitch: int) -> np.ndarray:
        """Get the sound data for an instrument at a given pitch. Returns a numpy
array."""
        return self._sounds[instrument][pitch]
```

---

loop_hijacker.py

```
from time import perf_counter_ns
from typing import Callable
import tkinter as tk

class LoopHijacker:
    """
    Turns inconsistent callback frequencies from the tkinter event loop
    into consistent ones for playback.
    """

    def __init__(
            self,
            root: tk.Tk,
            callback: Callable,
            tps: int = 20,
            lookahead_ticks: int = 5,
            repeat_ms: int = 25
    ):
        self.root = root
```

```
        self.callback = callback
        self.tps = tps
        self.lookahead_ticks = lookahead_ticks
        self.repeat_ms = repeat_ms
        self.enabled: bool = False
        self.reset()

    def reset(self):
        self.start_time = self.time_ms()
        self.next_tick = 0

    def enable(self):
        if not self.enabled:
            self.enabled = True
            self.reset()
            self.update()

    def disable(self):
        self.enabled = False

    def hijack_root(self):
        self.root.after(self.repeat_ms, self.update)

    def update(self):
        elapsed_time = self.time_ms() - self.start_time
        current_tick = self.ms_to_tick(elapsed_time) + self.lookahead_ticks
        while current_tick >= self.next_tick:
            self.callback()
            self.next_tick += 1
        if self.enabled: self.hijack_root()

    def time_ms(self) -> int:
        return int(perf_counter_ns() * 0.000001)

    def ms_to_tick(self, ms: int) -> int:
        return int(ms * self.tps * 0.001)
```

---

main.py

```
import tkinter as tk
import tkinter.ttk as ttk

from model import Model
from instrument_sounds import InstrumentSounds
from playback import Playback
from audio_exporter import AudioExporter

from pattern_list import PatternList
from sequencer import Sequencer
from top_frame import TopFrame
from bottom_frame import BottomFrame

def main():
    """Program flow starts here. Everything else is initialized from here."""

    model = Model()
```

```
    window = tk.Tk()
    window.title("project noteblock - new song")
    window.columnconfigure(0, weight=1)
    window.rowconfigure(1, weight=1)

    block_size = 2400
    sounds = InstrumentSounds(block_size=block_size)
    playback = Playback(model=model, window=window, sounds=sounds, block_size=bloc
k_size)
    audio_exporter = AudioExporter(model=model, sounds=sounds, block_size=block_si
ze)

    top_frame = TopFrame(window, model=model, playback=playback, audio_exporter=au
dio_exporter)

    main_frame = ttk.Frame(window)
    main_frame.columnconfigure(1, weight=1)
    main_frame.rowconfigure(0, weight=1)

    pattern_list = PatternList(main_frame, model=model)
    pattern_list.grid(column=0, row=0, padx=5, pady=5, sticky="nsew")

    sequencer = Sequencer(main_frame, model=model)
    sequencer.grid(column=1, row=0, padx=5, pady=5, sticky="nsew")

    bottom_frame = BottomFrame(window, model=model)

    top_frame.grid(column=0, row=0, sticky="ew")
    main_frame.grid(column=0, row=1, sticky="nsew")
    bottom_frame.grid(column=0, row=2, sticky="nsew")

    window.mainloop()

if __name__ == "__main__":
    main()
```

---

model.py

```
import json

from node import Node
from events import EventBus
from undo_manager import UndoManager
from node_editor import NodeEditor
from node_factory import NodeFactory

from song_config import SongConfig
from pattern import Pattern
from pattern_group import PatternGroup
from channel import Channel
from channel_group import ChannelGroup

class Model:
    """
    A wrapper and coordinator for song object classes and other helper classes.
    Also handles song init/load/save and song-wide edits.
    """
```

```python
    def __init__(self):
        self.init_components()
        self.init_tree()

    def init_components(self):
        self.uman = UndoManager()
        self.event_bus = EventBus()
        self.ed = NodeEditor(self.uman, self.event_bus)
        self.factory = NodeFactory()

    def init_tree(self):
        self.root = Node()
        self.song_config = SongConfig()
        self.pattern_group = PatternGroup()
        self.channel_group = ChannelGroup()
        self.root._add_child(self.song_config, 0, 0)
        self.root._add_child(self.pattern_group, 1, 1)
        self.root._add_child(self.channel_group, 2, 2)
        self.new_pattern()
        self.new_channel()
        self.event_bus.reset_ui()

    def from_dict(self, source: dict):
        self.song_config = self.factory.create_node(source["children"]["0"])
        self.pattern_group = self.factory.create_node(source["children"]["1"])
        self.channel_group = self.factory.create_node(source["children"]["2"],
            pattern_group=self.pattern_group)
        self.root = Node()
        self.root._add_child(self.song_config, 0, 0)
        self.root._add_child(self.pattern_group, 1, 1)
        self.root._add_child(self.channel_group, 2, 2)
        self.event_bus.reset_ui()

    def to_dict(self) -> dict:
        return self.root.to_dict()

    def from_file(self, filename: str):
        with open(filename, "r", encoding="utf-8") as file:
            self.from_dict(json.load(file))

    def to_file(self, filename: str):
        with open(filename, "w", encoding="utf-8") as file:
            json.dump(self.to_dict(), file)

    def new_pattern(self) -> Pattern:
        pattern_length = self.song_config.get_property("pattern_length")
        pattern = Pattern(pattern_length=pattern_length)
        self.ed.add_child(self.pattern_group, pattern)
        return pattern

    def new_channel(self) -> Channel:
        sequence_length = self.song_config.get_property("sequence_length")
        channel = Channel(pattern_group=self.pattern_group, sequence_length=sequen
ce_length)
        self.ed.add_child(self.channel_group, channel)
        return channel
```

```
    def change_pattern_length(self, new_length: int):
        self.uman.start_group()
        old_length: int = self.song_config.get_property("pattern_length")
        ratio = new_length / old_length
        for pattern in self.pattern_group.children_iterator():
            old_notes = pattern.get_property("notes")
            new_notes = [-1] * new_length
            for old_tick, note in reversed(list(enumerate(old_notes))):
                if note != -1:
                    new_tick = int(old_tick * ratio)
                    new_notes[new_tick] = note
            self.ed.set_property(pattern, "notes", new_notes)
        self.ed.set_property(self.song_config, "pattern_length", new_length)
        self.uman.end_group()

    def change_sequence_length(self, new_length: int):
        self.uman.start_group()
        old_length = self.song_config.get_property("sequence_length")
        for channel in self.channel_group.children_iterator():
            old_placements = channel.get_property("placements")
            if new_length > old_length:
                difference = new_length - old_length
                new_placements = old_placements + ([-1] * difference)
            else:
                new_placements = old_placements[:new_length]
            self.ed.set_property(channel, "placements", new_placements)
        self.ed.set_property(self.song_config, "sequence_length", new_length)
        self.uman.end_group()

    def remove_pattern(self, pattern: Pattern):
        self.uman.start_group()
        pattern_id = self.pattern_group.get_id_of_child(pattern)
        if pattern_id is None: return
        for channel in self.channel_group.children_iterator():
            old_placements = channel.get_property("placements")
            new_placements = [id if id != pattern_id else -1 for id in old_placeme
nts]
            self.ed.set_property(channel, "placements", new_placements)
        self.ed.remove_child(self.pattern_group, pattern)
        self.uman.end_group()
```

---

node.py

```
from copy import copy, deepcopy

class Node:
    """Base class of all song objects."""

    def __init__(self, *args, **kwargs):
        self.parent: Node | None = None
        self.properties = {}
        self.children: dict[int, Node] = {}
        self.child_order: list[int] = []

    def to_dict(self):
        return {
            "class": self.__class__.__name__,
```

```
            "properties": deepcopy(self.properties),
            "child_order": deepcopy(self.child_order),
            "children": {str(k): v.to_dict() for k, v in self.children.items()}
        }

    def get_property(self, key):
        # shouldn't need deepcopy here - nested data should be separated into node
s
        return copy(self.properties.get(key, None))

    def get_child_by_id(self, id: int):
        return self.children.get(id, None)

    def get_child_at_index(self, index: int):
        if 0 <= index < len(self.child_order):
            return self.get_child_by_id(self.child_order[index])
        return None

    def get_child_by_class(self, _class):
        for child in self.children.values():
            if isinstance(child, _class): return child

    def get_index_of_child(self, child: "Node"):
        child_id = self.get_id_of_child(child)
        if child_id is not None:
            return self.child_order.index(child_id)
        return None

    def get_id_of_child(self, child: "Node"):
        for k, v in self.children.items():
            if v == child:
                return k
        return None

    def is_root(self):
        return True if self.parent is None else False

    def next_available_id(self):
        return max(self.children.keys(), default=-1) + 1

    def children_iterator(self):
        for child_id in self.child_order:
            yield self.get_child_by_id(child_id)

    def children_count(self):
        return len(self.children)

    def _set_property(self, key, value):
        self.properties[key] = value

    def _add_child(self, child: "Node", id: int, index: int):
        self.children[id] = child
        if index is not None: self.child_order.insert(index, id) # allows no index
insertion for factory
        child.parent = self

    def _remove_child(self, child: "Node", id: int, index: int):
        del self.children[id]
```

```python
        del self.child_order[index]
        child.parent = None

    def _move_child(self, old_index: int, new_index: int):
        child_id = self.child_order.pop(old_index)
        self.child_order.insert(new_index, child_id)
```

---

node_actions.py

```python
from abc import ABC, abstractmethod
from node import Node
from events import EventBus

class Action(ABC):
    """
    Encapsulates a change to a node as an object.
    Subclasses that actually do things are found below.
    """

    @abstractmethod
    def __init__(self):
        ...

    @abstractmethod
    def perform(self):
        ...

    @abstractmethod
    def undo(self):
        ...

class SetPropertyAction(Action):
    def __init__(self, event_bus: EventBus, node: Node, key, old_value, new_value)
:
        self.event_bus = event_bus
        self.node = node
        self.key = key
        self.old_value = old_value
        self.new_value = new_value

    def perform(self):
        self.node._set_property(self.key, self.new_value)
        self.event_bus.node_property_set(self.node, self.key, self.old_value, self
.new_value)

    def undo(self):
        self.node._set_property(self.key, self.old_value)
        self.event_bus.node_property_set(self.node, self.key, self.new_value, self
.old_value)

class AddChildAction(Action):
    # assumes child does not have another parent
    def __init__(self, event_bus: EventBus, parent: Node, child: Node, id: int, in
dex: int):
        self.event_bus = event_bus
        self.parent = parent
        self.child = child
```

```
        self.id = id
        self.index = index

    def perform(self):
        self.parent._add_child(self.child, self.id, self.index)
        self.event_bus.node_child_added(self.parent, self.child, self.id, self.ind
ex)

    def undo(self):
        self.parent._remove_child(self.child, self.id, self.index)
        self.event_bus.node_child_removed(self.parent, self.child, self.id, self.i
ndex)

class RemoveChildAction(Action):
    def __init__(self, event_bus: EventBus, parent: Node, child: Node, id: int, in
dex: int):
        self.event_bus = event_bus
        self.parent = parent
        self.child = child
        self.id = id
        self.index = index

    def perform(self):
        self.parent._remove_child(self.child, self.id, self.index)
        self.event_bus.node_child_removed(self.parent, self.child, self.id, self.i
ndex)

    def undo(self):
        self.parent._add_child(self.child, self.id, self.index)
        self.event_bus.node_child_added(self.parent, self.child, self.id, self.ind
ex)

class MoveChildAction(Action):
    def __init__(self, event_bus: EventBus, parent: Node, old_index: int, new_inde
x: int):
        self.event_bus = event_bus
        self.parent = parent
        self.old_index = old_index
        self.new_index = new_index

    def perform(self):
        self.parent._move_child(self.old_index, self.new_index)
        self.event_bus.node_child_moved(self.parent, self.old_index, self.new_inde
x)

    def undo(self):
        self.parent._move_child(self.new_index, self.old_index)
        self.event_bus.node_child_moved(self.parent, self.new_index, self.old_inde
x)
```

node_editor.py

```
from node import Node
from events import EventBus
from node_actions import AddChildAction, RemoveChildAction, MoveChildAction, SetPr
opertyAction
from undo_manager import UndoManager
```

```python
class NodeEditor:
    """
    Provides easier methods for editing a song object tree,
    including special case handling and undo management.
    """

    def __init__(self, uman: UndoManager, event_bus: EventBus):
        self.uman = uman
        self.event_bus = event_bus

    def set_property(self, node: Node, key, value):
        old_value = node.get_property(key)
        self.uman.perform(SetPropertyAction(
            self.event_bus,
            node,
            key,
            old_value,
            value
        ))

    def toggle_bool(self, node: Node, key):
        old_value = node.get_property(key)
        if not isinstance(old_value, bool):
            return
        new_value = False if old_value else True
        self.uman.perform(SetPropertyAction(
            self.event_bus,
            node,
            key,
            old_value,
            new_value
        ))

    def remove_child(self, parent: Node, child: Node):
        if child.parent is not parent: return
        child_id = parent.get_id_of_child(child)
        child_index = parent.get_index_of_child(child)
        if child_id is None or child_index is None: return
        self.uman.perform(RemoveChildAction(
            self.event_bus,
            parent,
            child,
            child_id,
            child_index
        ))

    def remove_child_with_id(self, parent: Node, child_id: int):
        child = parent.get_child_by_id(child_id)
        child_index = parent.get_index_of_child(child)
        if child is None or child_index is None: return
        self.uman.perform(RemoveChildAction(
            self.event_bus,
            parent,
            child,
            child_id,
            child_index
        ))
```

```python
def remove_child_at_index(self, parent: Node, child_index: int):
    child = parent.get_child_at_index(child_index)
    child_id = parent.get_id_of_child(child)
    if child is None or child_id is None: return
    self.uman.perform(RemoveChildAction(
        self.event_bus,
        parent,
        child,
        child_id,
        child_index
    ))

def add_child(self, parent: Node, child: Node):
    if child.parent is not None: self.remove_child(parent, child)
    child_id = parent.next_available_id()
    child_index = len(parent.child_order)
    self.uman.perform(AddChildAction(
        self.event_bus,
        parent,
        child,
        child_id,
        child_index
    ))

def add_child_with_id(self, parent: Node, child: Node, child_id: int):
    if child_id in parent.children.keys(): return
    if child.parent is not None: self.remove_child(parent, child)
    child_index = len(parent.child_order)
    self.uman.perform(AddChildAction(
        self.event_bus,
        parent,
        child,
        child_id,
        child_index
    ))

def add_child_at_index(self, parent: Node, child: Node, child_index: int):
    if child.parent is not None: self.remove_child(parent, child)
    child_id = parent.next_available_id()
    self.uman.perform(AddChildAction(
        self.event_bus,
        parent,
        child,
        child_id,
        child_index
    ))

def move_child(self, parent: Node, old_index: int, new_index: int):
    children_count = parent.children_count()
    if 0 <= old_index < children_count and 0 <= new_index < children_count:
        self.uman.perform(MoveChildAction(
            self.event_bus,
            parent,
            old_index,
            new_index
        ))
```

node_factory.py

```python
from copy import deepcopy
from typing import Type

from node import Node
from song_config import SongConfig
from pattern import Pattern
from pattern_group import PatternGroup
from channel import Channel
from channel_group import ChannelGroup

from effect_dummy import EffectDummy
from effect_delay import EffectDelay
# TODO: import Node subclasses here

class NodeFactory:
    """
    Creates a song object from its class name and attributes.
    Used to reproduce a song object from a dictionary.
    """

    node_classes: dict[str, Type[Node]] = {
        "Node": Node,
        "SongConfig": SongConfig,
        "Pattern": Pattern,
        "PatternGroup": PatternGroup,
        "Channel": Channel,
        "ChannelGroup": ChannelGroup,
        "EffectDummy": EffectDummy,
        "EffectDelay": EffectDelay,
        # TODO: add entries for Node subclasses here
    }

    def create_node(self, source: dict, **kwargs):
        node_class = self.node_classes.get(source["class"], None)
        if node_class is None: return None
        node = node_class(**kwargs)
        node.properties = deepcopy(source["properties"])
        node.child_order = deepcopy(source["child_order"])
        for k, v in source["children"].items():
            child = self.create_node(v, **kwargs)
            node._add_child(child, id=int(k), index=None)
        return node
```

---

note.py

```python
from dataclasses import dataclass

@dataclass(slots=True)
class Note:
    """
    Represents one note played by a note block.
    Generating and processing these is the basis of playback.
    """

    instrument: int = 0
    pitch: int = 0
```

```python
    volume: float = 1 # 0 to 1
    pan: float = 0 # -1 to 1

    def apply_volume_and_pan(self, volume: float = 1.0, pan: float = 0.0):
        new_volume = min(self.volume * volume, 1)
        new_pan = max(min(self.pan + pan, 1), -1)
        return Note(
            instrument=self.instrument,
            pitch=self.pitch,
            volume=new_volume,
            pan=new_pan
        )
```

---

pattern.py

```python
from node import Node

class Pattern(Node):
    """Song object - contains a chronological list of notes."""

    def __init__(self, *args, pattern_length: int = 16, **kwargs):
        super().__init__(*args, **kwargs)
        self._set_property("name", "pattern name")
        self._set_property("colour", "gray75")
        self._set_property("notes", [-1] * pattern_length)

    def get_notes(self, pat_tick: int) -> list[int]:
        return [self.get_property("notes")[pat_tick]]
```

---

pattern_group.py

```python
from node import Node

class PatternGroup(Node):
    """Song object - holds all of a song's patterns."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

---

pattern_list.py

```python
import tkinter as tk
import tkinter.ttk as ttk
from tkinter import dnd

from node import Node
from pattern import Pattern
from events import Listener
from model import Model

class PatternList(Listener, ttk.Frame):
    """
    UI component - shows a list of patterns that
    can be created/reordered/deleted and dragged into the sequencer.
    """
```

```
    def __init__(self, parent, *args, model: Model, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.model = model
        self.columnconfigure(0, weight=1)
        self.rowconfigure(0, weight=1)

        self.name_width: int = 20

        self.init_ui()
        self.model.event_bus.add_listener(self)
        self.update_ui()

    def destroy(self, *args, **kwargs):
        self.model.event_bus.remove_listener(self)
        super().destroy(*args, **kwargs)

    def node_property_set(self, node: Node, key, old_value, new_value):
        if isinstance(node, Pattern) and key != "notes":
            self.update_ui()

    def node_child_added(self, parent: Node, child: Node, id: int, index: int):
        if parent is self.model.pattern_group:
            self.update_ui()

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        if parent is self.model.pattern_group:
            self.update_ui()

    def node_child_moved(self, parent: Node, old_index: int, new_index: int):
        if parent is self.model.pattern_group:
            self.update_ui()

    def reset_ui(self):
        self.update_ui()

    def init_ui(self):
        self.canvas = tk.Canvas(self)
        self.canvas.grid(column=0, row=0, sticky="ns")

        self.internal_frame = ttk.Frame(self.canvas)
        self.canvas.create_window(0, 0, anchor="nw", window=self.internal_frame)

        # NOTE: this is the magic line
        # resizing canvas to frame here instead of in update_ui() avoids the
        # problem of having 1 pixel width/height at init
        self.internal_frame.bind(
            "<Configure>",
            lambda e: self.canvas.configure(width=self.internal_frame.winfo_reqwid
th(),
                                            scrollregion=(0, 0, 0, self.internal_f
rame.winfo_reqheight()))
        )

        self.scrollbar = ttk.Scrollbar(self, orient=tk.VERTICAL)
        self.canvas.configure(yscrollcommand=self.scrollbar.set)
        self.scrollbar["command"] = self.canvas.yview
        self.scrollbar.grid(column=1, row=0, sticky="ns")
```

```
        self.btn_add_pattern = ttk.Button(
            self,
            text="+ add pattern",
            command=self.model.new_pattern
        )
        self.btn_add_pattern.grid(column=0, row=1)

    def update_ui(self):
        for child in self.internal_frame.winfo_children():
            child.destroy()

        for index, pattern in enumerate(self.model.pattern_group.children_iterator
()):
            pattern_name = pattern.get_property("name")
            pattern_colour = pattern.get_property("colour")
            pattern_label = ttk.Label(
                self.internal_frame,
                text=pattern_name,
                width=self.name_width,
                background=pattern_colour,
            )
            pattern_label.pattern = pattern # cheeky monkey patch
            pattern_label.dnd_end = lambda t, e: ... # empty function

            # cursed hack around Python closures in lambda functions
            pattern_label.bind("<ButtonPress-1>", lambda e, p=pattern: self.model.
event_bus.node_selected(p))
            pattern_label.bind("<ButtonPress-1>", lambda e, p=pattern_label: dnd.d
nd_start(p, e), add=True)

            btn_move_up = ttk.Button(
                self.internal_frame,
                text="▲",
                width=2,
                command=lambda i=index: self.move_pattern(i, -1)
            )
            btn_move_down = ttk.Button(
                self.internal_frame,
                text="▼",
                width=2,
                command=lambda i=index: self.move_pattern(i, 1)
            )
            btn_delete = ttk.Button(
                self.internal_frame,
                text="🗑",
                width=2,
                command=lambda p=pattern: self.model.remove_pattern(p)
            )

            pattern_label.grid(column=0, row=index, ipadx=5, ipady=5, padx=2, pady
=2)
            btn_move_up.grid(column=1, row=index, padx=2, pady=2)
            btn_move_down.grid(column=2, row=index, padx=2, pady=2)
            btn_delete.grid(column=3, row=index, padx=2, pady=2)

    def move_pattern(self, old_index: int, delta: int):
```

```
        new_index = old_index + delta
        self.model.ed.move_child(self.model.pattern_group, old_index, new_index)
```

---

pattern_settings.py

```
import tkinter as tk
import tkinter.ttk as ttk
from tkinter import colorchooser

from node import Node
from events import Listener
from model import Model

from pattern import Pattern
from piano_roll_canvas import PianoRollCanvas

class PatternSettings(Listener, ttk.Frame):
    """UI component - shows a pattern's settings above the piano roll."""

    padding = {"padx": 2, "pady": 2}

    def __init__(self, parent, *args, model: Model, canvas: PianoRollCanvas, **kwa
rgs):
        super().__init__(parent, *args, **kwargs)
        self.model = model
        self.canvas = canvas
        self.pattern: Pattern | None = None

        self.init_ui()
        self.model.event_bus.add_listener(self)
        self.update_ui()

    def destroy(self, *args, **kwargs):
        self.model.event_bus.remove_listener(self)
        super().destroy(*args, **kwargs)

    def node_property_set(self, node: Node, key, old_value, new_value):
        if node is self.pattern and key != "notes":
            self.update_ui()

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        if parent is self.model.pattern_group and child is self.pattern:
            self.pattern = None
            self.update_ui()

    def node_selected(self, node: Node):
        if isinstance(node, Pattern):
            self.pattern = node
            self.update_ui()

    def reset_ui(self):
        self.pattern = None
        self.update_ui()

    def choose_colour(self, event: tk.Event):
        if self.pattern is not None:
            colour = colorchooser.askcolor()[1]
```

```
            if colour is not None: self.model.ed.set_property(self.pattern, "colou
r", colour)

    def init_ui(self):
        self.lbl_colour = ttk.Label(self, width=3)
        self.lbl_colour.bind(
            "<ButtonPress-1>",
            self.choose_colour
        )

        self.var_name = tk.StringVar(self)
        self.inp_name = ttk.Entry(
            self,
            width=20,
            textvariable=self.var_name,
        )
        self.inp_name.bind(
            "<Return>",
            lambda e: self.model.ed.set_property(self.pattern, "name", self.var_na
me.get())
        )

        self.btn_zoom_in = ttk.Button(
            self,
            text="+",
            width=3,
            command=lambda: self.canvas.zoom(1.25)
        )
        self.btn_zoom_out = ttk.Button(
            self,
            text="-",
            width=3,
            command=lambda: self.canvas.zoom(0.8)
        )

        self.btn_delete = ttk.Button(
            self,
            text="🗑",
            width=3,
            command=lambda: self.model.remove_pattern(self.pattern)
        )

        self.lbl_colour.grid(column=0, row=0, **self.padding)
        self.inp_name.grid(column=1, row=0, **self.padding)
        self.btn_zoom_in.grid(column=2, row=0, **self.padding)
        self.btn_zoom_out.grid(column=3, row=0, **self.padding)
        self.btn_delete.grid(column=5, row=0, sticky="e", **self.padding)

    def update_ui(self):
        if self.pattern is None:
            self.set_all_states(["disabled"])
        else:
            self.set_all_states(["!disabled"])
            self.lbl_colour.config(background=self.pattern.get_property("colour"))
            self.var_name.set(self.pattern.get_property("name"))

    def set_all_states(self, statespec: list[str]):
        for component in (self.inp_name, self.btn_zoom_in, self.btn_zoom_out, self
```

```
.btn_delete):
            component.state(statespec)
```

---

piano_notes_canvas.py

```
import math
import tkinter as tk

class PianoNotesCanvas(tk.Canvas):
    """UI component - displays a piano's keys next to the piano roll for reference
."""

    def __init__(self, parent, *args, **kwargs):
        self.canvas_height: int = 300
        self.note_width: int = 30

        # drawing constants
        self.non_neg_pitch_count: int = 25
        self.negative_pitch_count: int = 2
        self.pitch_count: int = self.non_neg_pitch_count + self.negative_pitch_cou
nt
        self.draw_no_note: bool = False # for debugging idk
        self.note_height: float = self.canvas_height / self.pitch_count

        # colour constants
        self.bg_colour: str = "white"
        self.guidebar_colour: str = "black"
        self.no_note_bar_colour: str = "gray50"

        super().__init__(
            parent, *args,
            height=self.canvas_height,
            width=self.note_width,
            highlightthickness=0,
            bg=self.bg_colour,
            **kwargs
        )

        for note in self.black_notes(self.pitch_count):
            self.draw_note(note, 0, length=1, fill=self.guidebar_colour, outline="
")
        self.draw_note(-1, 0, length=1, fill=self.no_note_bar_colour, outline="")

    def draw_note(self, note: int, tick: int, length: int, **kwargs) -> int:
        """Draws a note on the canvas."""
        self.create_rectangle(
            tick * self.note_width,
            self.note_height * (self.non_neg_pitch_count - 1 - note),
            (tick + length) * self.note_width,
            self.note_height * (self.non_neg_pitch_count - note),
            **kwargs
        )

    def black_notes(self, pitch_count: int) -> list[int]:
        """Helper function which returns a list of black notes within the given pi
tch range."""
        result = []
```

```
        octave_black_notes = [0, 2, 4, 7, 9]
        octave_count = math.ceil(pitch_count / 12)
        for octave in range(octave_count):
            result.extend(n + 12*octave for n in octave_black_notes)
        result = filter(lambda n: n < pitch_count, result)
        return result
```

---

piano_roll.py

```
import tkinter as tk
import tkinter.ttk as ttk

from model import Model
from piano_roll_canvas import PianoRollCanvas
from piano_notes_canvas import PianoNotesCanvas
from pattern_settings import PatternSettings

class PianoRoll(ttk.Frame):
    """UI component - a scrollable, editable pattern display."""

    def __init__(self, parent, *args, model: Model, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.model = model

        self.piano_roll_canvas = PianoRollCanvas(self, model=self.model)
        self.piano_notes_canvas = PianoNotesCanvas(self)
        self.pattern_settings = PatternSettings(self, model=model, canvas=self.pia
no_roll_canvas)

        self.scrollbar = ttk.Scrollbar(self, orient=tk.HORIZONTAL)
        self.piano_roll_canvas.configure(xscrollcommand=self.scrollbar.set)
        self.scrollbar["command"] = self.piano_roll_canvas.xview

        self.columnconfigure(1, weight=1)
        self.rowconfigure(1, weight=1)
        self.pattern_settings.grid(column=0, row=0, columnspan=2, sticky="ew")
        self.piano_notes_canvas.grid(column=0, row=1)
        self.piano_roll_canvas.grid(column=1, row=1, sticky="ew")
        self.scrollbar.grid(column=1, row=2, sticky="ew")
```

---

piano_roll_canvas.py

```
import math
import tkinter as tk

from node import Node
from events import Listener
from model import Model

from pattern import Pattern

class PianoRollCanvas(Listener, tk.Canvas):
    """UI component - the main renderer of the piano roll."""

    def __init__(self, parent, *args, model: Model, **kwargs):
        self.canvas_height: int = 300
```

```
        self.note_width: float = 20
        self.model = model
        self.pattern: Pattern | None = None

        # drawing constants
        self.non_neg_pitch_count: int = 25
        self.negative_pitch_count: int = 2
        self.pitch_count: int = self.non_neg_pitch_count + self.negative_pitch_cou
nt
        self.draw_no_note: bool = False # for debugging idk
        self.note_height: float = self.canvas_height / self.pitch_count

        # colour constants
        self.bg_colour: str = "gray75"
        self.guidebar_colour: str = "gray70"
        self.guideline_colour: str = "gray65"
        self.no_note_bar_colour: str = "gray65"

        super().__init__(
            parent, *args,
            height=self.canvas_height,
            scrollregion=(0, 0, self.target_canvas_length(), self.canvas_height),
            highlightthickness=0,
            bg=self.bg_colour,
            **kwargs
        )

        self.model.event_bus.add_listener(self)
        self.bind("<ButtonPress-1>", self.set_note) # left click sets note
        self.bind("<ButtonPress-3>", self.delete_note) # right click deletes note

        self.draw_everything()

    def destroy(self, *args, **kwargs):
        self.model.event_bus.remove_listener(self)
        super().destroy(*args, **kwargs)

    def node_property_set(self, node: Node, key, old_value, new_value):
        if node is self.pattern:
            self.draw_everything()

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        if parent is self.model.pattern_group and child is self.pattern:
            self.pattern = None
            self.draw_everything()

    def node_selected(self, node: Node):
        if isinstance(node, Pattern):
            self.pattern = node
            self.draw_everything()

    def reset_ui(self):
        self.pattern = None
        self.draw_everything()

    def zoom(self, zoom_factor: float):
        """Stretches the piano roll horizontally by the given factor."""
        left_fraction = self.xview()[0]
```

```
        self.note_width *= zoom_factor
        self.configure(scrollregion=(0, 0, self.target_canvas_length(), self.canva
s_height))
        self.xview_moveto(left_fraction)
        self.draw_everything()

    def draw_everything(self):
        """Clears and redraws the canvas."""
        self.delete("all")
        # make sure the canvas is the right length if pattern length has changed!
        self.configure(scrollregion=(0, 0, self.target_canvas_length(), self.canva
s_height))
        self.draw_guide_bars()
        self.draw_guide_lines()
        self.draw_pattern_notes()

    def draw_guide_bars(self):
        """Draws the horizontal guide bars."""
        length = self.pattern_length()
        for note in self.black_notes(self.pitch_count):
            self.draw_note(note, 0, length=length, fill=self.guidebar_colour, outl
ine="")
        self.draw_note(-1, 0, length=length, fill=self.no_note_bar_colour, outline
="")

    def draw_guide_lines(self):
        """Draws the vertical guide lines."""
        for i in range(self.pattern_length()):
            self.create_line(
                i * self.note_width,
                0,
                i * self.note_width,
                self.canvas_height,
                fill=self.guideline_colour,
                width=0
            )

    def draw_pattern_notes(self):
        """Draws all notes in the pattern."""
        if self.pattern is not None:
            for tick, note in enumerate(self.pattern.get_property("notes")):
                if note != -1 or self.draw_no_note:
                    self.draw_note(note, tick, length=1, fill=self.pattern.get_pro
perty("colour"))

    def draw_note(self, note: int, tick: int, length: int, **kwargs) -> int:
        """Draws a note on the canvas."""
        self.create_rectangle(
            tick * self.note_width,
            self.note_height * (self.non_neg_pitch_count - 1 - note),
            (tick + length) * self.note_width,
            self.note_height * (self.non_neg_pitch_count - note),
            **kwargs
        )

    def get_note_at_coords(self, x: int, y: int) -> tuple[int, int]:
        """Translates widget-relative coordinates to the corresponding note and ti
ck."""
        canvas_x = self.canvasx(x)
```

```python
        canvas_y = self.canvasy(y)
        tick = int(canvas_x // self.note_width)
        note = int(self.non_neg_pitch_count - 1 - (canvas_y // self.note_height))
        return note, tick

    def delete_note(self, event: tk.Event):
        """Deletes a note at the given event coordinates."""
        if self.pattern is not None:
            note, tick = self.get_note_at_coords(event.x, event.y)
            if tick < self.pattern_length() and self.pattern.get_property("notes")
[tick] == note:
                notes = self.pattern.get_property("notes")
                notes[tick] = -1
                self.model.ed.set_property(self.pattern, "notes", notes)

    def set_note(self, event: tk.Event):
        """Sets a note at the given event coordinates. If a note is already at tha
t tick it is replaced."""
        if self.pattern is not None:
            note, tick = self.get_note_at_coords(event.x, event.y)
            if tick < self.pattern_length():
                notes = self.pattern.get_property("notes")
                notes[tick] = note
                self.model.ed.set_property(self.pattern, "notes", notes)

    def black_notes(self, pitch_count: int) -> list[int]:
        """Helper function which returns a list of black notes within the given pi
tch range."""
        result = []
        octave_black_notes = [0, 2, 4, 7, 9]
        octave_count = math.ceil(pitch_count / 12)
        for octave in range(octave_count):
            result.extend(n + 12*octave for n in octave_black_notes)
        result = filter(lambda n: n < pitch_count, result)
        return result

    def target_canvas_length(self) -> int:
        """Helper function to calculate how long the canvas should be."""
        return self.note_width * self.pattern_length()

    def pattern_length(self):
        if self.pattern is not None:
            return len(self.pattern.get_property("notes"))
        else:
            return 0
```

---

placement_display.py

```python
import tkinter as tk
import tkinter.ttk as ttk

from node import Node
from events import EventBus, Listener
from node_editor import NodeEditor
from model import Model

from pattern import Pattern
```

```python
from pattern_group import PatternGroup
from channel import Channel
from channel_group import ChannelGroup

class PlacementDisplay(Listener, tk.Canvas):
    """
    UI component - the main renderer of the sequencer.
    Shows pattern placements on their respective channels.
    """

    def __init__(self, parent, *args, model: Model, **kwargs):
        self.model = model
        self.selected_bar: int = 0
        self.playing_bar: int = -1

        # actual config
        self.pattern_width: float = 60
        self.pattern_height: float = 60

        # internal variables that get recalculated
        self.channel_count: int = 0
        self.placement_count: int = 0
        self.canvas_height: float = 0
        self.canvas_width: float = 0

        self.bg_colour: str = "gray75"
        self.guidebar_colour: str = "gray70"
        self.guideline_colour: str = "gray65"
        self.selected_bar_colour: str = "red"
        self.playing_bar_colour: str = "green"

        super().__init__(
            parent,
            *args,
            scrollregion=(0, 0, 0, 0),
            highlightthickness=0,
            bg=self.bg_colour,
            **kwargs
        )
        self.bind("<ButtonPress-1>", self.select_things)
        self.bind("<ButtonPress-3>", self.delete_placement)

        self.model.event_bus.add_listener(self)
        self.draw_everything()

    def destroy(self, *args, **kwargs):
        self.model.event_bus.remove_listener(self)
        super().destroy(*args, **kwargs)

    def node_property_set(self, node: Node, key, old_value, new_value):
        if isinstance(node, Pattern) and key == "colour":
            self.draw_placements()
        elif isinstance(node, Channel) and key == "placements":
            self.draw_placements()
        elif node is self.model.song_config and key == "sequence_length":
            self.draw_everything()

    def node_child_added(self, parent: Node, child: Node, id: int, index: int):
```

```python
        if parent is self.model.channel_group:
            self.draw_everything()

    def node_child_removed(self, parent: Node, child: Node, id: int, index: int):
        if parent is self.model.channel_group:
            self.draw_everything()

    def node_child_moved(self, parent: Node, old_index: int, new_index: int):
        if parent is self.model.channel_group:
            self.draw_everything()

    def bar_selected(self, bar: int):
        self.selected_bar = bar
        self.draw_selected_bar_line()

    def bar_playing(self, bar: int):
        self.playing_bar = bar
        self.draw_playing_bar_line()

    def reset_ui(self):
        self.draw_everything()

    # NOTE: apparently all of the following are needed to use Tkinter's drag and d
rop library
    # maybe I should have written my own...

    def dnd_accept(self, source, event):
        return self

    def dnd_enter(self, source, event):
        ...

    def dnd_motion(self, source, event):
        ...

    def dnd_leave(self, source, event):
        ...

    def dnd_commit(self, source, event: tk.Event):
        pattern = getattr(source, "pattern", None)
        if isinstance(pattern, Pattern):
            pattern_id = self.model.pattern_group.get_id_of_child(pattern)
            bar, channel, _ = self.get_everything_at_coords(
                event.x_root - self.winfo_rootx(),
                event.y_root - self.winfo_rooty()
            )
            if channel is not None:
                placements_copy = channel.get_property("placements")[:]
                placements_copy[bar] = pattern_id
                self.model.ed.set_property(channel, "placements", placements_copy)

    def draw_everything(self):
        self.recalculate_dimensions()
        self.configure_canvas()
        self.draw_placements()
        self.draw_selected_bar_line()
        self.draw_playing_bar_line()
```

```
    def recalculate_dimensions(self):
        self.channel_count = self.model.channel_group.children_count()
        self.placement_count = self.model.song_config.get_property("sequence_lengt
h")
        self.canvas_height = self.channel_count * self.pattern_height
        self.canvas_width = self.placement_count * self.pattern_width

    def configure_canvas(self):
        self.delete("all")
        self.configure(scrollregion=(0, 0, self.canvas_width, self.canvas_height))

        # guide bars
        for channel_number in range(0, self.channel_count, 2):
            self.draw_pattern(
                channel=channel_number,
                bar=0,
                length=self.placement_count,
                fill=self.guidebar_colour,
                outline=""
            )

        # guide lines
        for bar in range(self.placement_count):
            self.create_line(
                bar * self.pattern_width,
                0,
                bar * self.pattern_width,
                self.canvas_height,
                fill=self.guideline_colour,
                width=0
            )

    def draw_selected_bar_line(self):
        # selected bar line
        self.delete("selected_bar_line")
        if 0 <= self.selected_bar < self.placement_count:
            self.create_line(
                self.selected_bar * self.pattern_width,
                0,
                self.selected_bar * self.pattern_width,
                self.canvas_height,
                fill=self.selected_bar_colour,
                width=0,
                tags="selected_bar_line"
            )

    def draw_playing_bar_line(self):
        # playing bar line
        self.delete("playing_bar_line")
        if 0 <= self.playing_bar < self.placement_count:
            self.create_line(
                self.playing_bar * self.pattern_width,
                0,
                self.playing_bar * self.pattern_width,
                self.canvas_height,
                fill=self.playing_bar_colour,
                width=0,
                tags="playing_bar_line"
            )
```

```python
    def draw_placements(self):
        self.delete("placements")
        for channel_number, channel in enumerate(self.model.channel_group.children
_iterator()):
            for bar, pattern_id in enumerate(channel.get_property("placements")):
                if pattern_id == -1: continue # no placement here
                pattern = self.model.pattern_group.get_child_by_id(pattern_id)
                if pattern is None: continue
                pattern_colour = pattern.get_property("colour")
                self.draw_pattern(channel=channel_number, bar=bar, length=1, fill=
pattern_colour, tags="placements")
        self.tag_raise("selected_bar_line")
        self.tag_raise("playing_bar_line")

    def draw_pattern(self, channel: int, bar: int, length: int, **kwargs) -> int:
        """Draws a note on the canvas."""
        self.create_rectangle(
            self.pattern_width * bar,
            self.pattern_height * channel,
            self.pattern_width * (bar + length),
            self.pattern_height * (channel + 1),
            **kwargs
        )

    def select_things(self, event: tk.Event):
        bar, channel, pattern = self.get_everything_at_coords(event.x, event.y)
        if channel is not None: self.model.event_bus.node_selected(channel)
        if pattern is not None: self.model.event_bus.node_selected(pattern)

    def delete_placement(self, event: tk.Event):
        bar, channel, pattern = self.get_everything_at_coords(event.x, event.y)
        if pattern is not None:
            placements_copy = channel.get_property("placements")[:]
            placements_copy[bar] = -1
            self.model.ed.set_property(channel, "placements", placements_copy)

    def get_bar_at_coords(self, x: int, y: int) -> tuple[int, int]:
        canvas_x = self.canvasx(x)
        canvas_y = self.canvasy(y)
        bar = int(canvas_x // self.pattern_width)
        channel_index = int(canvas_y // self.pattern_height)
        return channel_index, bar

    def get_everything_at_coords(self, x: int, y: int):
        channel_index, bar = self.get_bar_at_coords(x, y)
        channel = self.model.channel_group.get_child_at_index(channel_index)
        if channel is None: return (bar, channel, None)
        pattern_id = channel.get_property("placements")[bar]
        pattern = self.model.pattern_group.get_child_by_id(pattern_id)
        return (bar, channel, pattern)
```

---

playback.py

```python
import queue
import numpy as np
from events import Listener
```

```python
from model import Model
from tick_manager import TickManager
from loop_hijacker import LoopHijacker
from instrument_sounds import InstrumentSounds
from audio_generator import AudioGenerator
from audio_player import AudioPlayer

class Playback(Listener):
    """Coordinates real-time playback of a song."""

    def __init__(self, model: Model, window, sounds: InstrumentSounds, block_size:
int = 2400):
        self.model = model
        self.model.event_bus.add_listener(self)
        self.start_bar: int = 0

        self.tick_manager = TickManager(model=model)
        self.loop_hijacker = LoopHijacker(
            root=window,
            callback=self.tick,
            tps=20,
            lookahead_ticks=3,
            repeat_ms=50
        )

        self.audio_generator = AudioGenerator(block_size=block_size, sounds=sounds
)
        self.audio_queue = queue.Queue(maxsize=10) # a bit arbitrary

        self.audio_player = AudioPlayer(self.audio_queue)
        self.loop_hijacker.enable()

    def bar_selected(self, bar: int):
        self.start_bar = bar
        if not self.tick_manager.sequence_enabled:
            self.tick_manager.set_tick(bar_number=bar, pat_tick=0)

    def play(self):
        self.tick_manager.enable_sequence()

    def pause(self):
        self.tick_manager.disable_sequence()

    def stop(self):
        self.pause()
        self.tick_manager.set_tick(bar_number=self.start_bar, pat_tick=0)

    def tick(self):
        next_tick = self.tick_manager.next_tick()
        notes = self.model.channel_group.tick(*next_tick)
        audio_block: np.ndarray = self.audio_generator.tick(notes)
        self.audio_queue.put(audio_block)
```

---

sequencer.py

```python
import tkinter as tk
import tkinter.ttk as ttk
```

```
from node import Node
from events import Listener
from model import Model

from channel_header_canvas import ChannelHeaderCanvas
from placement_display import PlacementDisplay
from bar_display import BarDisplay

class Sequencer(Listener, ttk.Frame):
    """UI component - facilitates song arrangement based on patterns and channels.
"""

    def __init__(self, parent, *args, model: Model, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.columnconfigure(0, weight=1)
        self.rowconfigure(1, weight=1)
        self.model = model

        self.placement_display = PlacementDisplay(self, model=self.model)
        self.channel_header_canvas = ChannelHeaderCanvas(self, model=self.model)
        self.bar_display = BarDisplay(self, model=self.model)

        self.buttons_frame = ttk.Frame(self)
        self.btn_loop = ttk.Button(
            self.buttons_frame,
            text="loop",
            command=lambda: self.model.ed.toggle_bool(self.model.song_config, "loo
p_enabled")
        )
        self.btn_loop.state(["pressed"] if self.model.song_config.get_property("lo
op_enabled") else ["!pressed"])
        self.btn_loop.grid(column=0, row=0)
        self.btn_add_channel = ttk.Button(
            self.buttons_frame,
            text="+ add channel",
            command=self.model.new_channel
        )
        self.btn_add_channel.grid(column=1, row=0)

        def xview_both_canvases(*args):
            self.placement_display.xview(*args)
            self.bar_display.xview(*args)

        self.horizontal_scroll = ttk.Scrollbar(self, orient=tk.HORIZONTAL)
        self.placement_display.configure(xscrollcommand=self.horizontal_scroll.set
)
        self.horizontal_scroll["command"] = xview_both_canvases

        def yview_both_canvases(*args):
            self.placement_display.yview(*args)
            self.channel_header_canvas.yview(*args)

        self.vertical_scroll = ttk.Scrollbar(self, orient=tk.VERTICAL)
        self.placement_display.configure(yscrollcommand=self.vertical_scroll.set)
        self.vertical_scroll["command"] = yview_both_canvases

        self.bar_display.grid(column=0, row=0, sticky="ew")
```

```
        self.placement_display.grid(column=0, row=1, sticky="nsew")
        self.channel_header_canvas.grid(column=1, row=1, sticky="ns")
        self.buttons_frame.grid(column=1, row=0, sticky="ew")
        self.horizontal_scroll.grid(column=0, row=2, sticky="ew")
        self.vertical_scroll.grid(column=2, row=1, sticky="ns")

        self.model.event_bus.add_listener(self)

    def destroy(self):
        self.model.event_bus.remove_listener(self)
        super().destroy()

    def node_property_set(self, node: Node, key, old_value, new_value):
        if node is self.model.song_config and key == "loop_enabled":
            self.btn_loop.state(["pressed"] if new_value else ["!pressed"])
```

---

song_config.py

```
from node import Node

class SongConfig(Node):
    """Song object - holds song-wide properties."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._set_property("name", "song name")
        self._set_property("pattern_length", 16)
        self._set_property("sequence_length", 20)
        self._set_property("loop_enabled", False)
        self._set_property("loop_start", 0)
        self._set_property("loop_end", 4)
```

---

song_settings.py

```
import tkinter as tk
import tkinter.ttk as ttk

from model import Model

class SongSettings(tk.Toplevel):
    """UI component - popup for editing song-wide properties."""

    def __init__(self, parent, model: Model, **kwargs):
        super().__init__(parent, **kwargs)
        self.model = model
        self.title("song settings")

        self.lbl_song_name = ttk.Label(self, text="song name:")
        self.lbl_pattern_length = ttk.Label(self, text="pattern length:")
        self.lbl_sequence_length = ttk.Label(self, text="sequence length:")

        self.var_song_name = tk.StringVar(self, value=self.model.song_config.get_p
roperty("name"))
        self.var_pattern_length = tk.IntVar(self, value=self.model.song_config.get
_property("pattern_length"))
        self.var_sequence_length = tk.IntVar(self, value=self.model.song_config.ge
```

```
t_property("sequence_length"))

        self.inp_song_name = ttk.Entry(self, textvariable=self.var_song_name, widt
h=25)
        self.inp_pattern_length = ttk.Spinbox(self, textvariable=self.var_pattern_
length, width=10)
        self.inp_sequence_length = ttk.Spinbox(self, textvariable=self.var_sequenc
e_length, width=10)

        self.btn_ok = ttk.Button(self, text="ok", command=self.save_settings)
        self.btn_cancel = ttk.Button(self, text="cancel", command=self.destroy)

        self.lbl_song_name.grid(column=0, row=0, sticky="e")
        self.lbl_pattern_length.grid(column=0, row=1, sticky="e")
        self.lbl_sequence_length.grid(column=0, row=2, sticky="e")
        self.inp_song_name.grid(column=1, row=0, sticky="w")
        self.inp_pattern_length.grid(column=1, row=1, sticky="w")
        self.inp_sequence_length.grid(column=1, row=2, sticky="w")
        self.btn_cancel.grid(column=0, row=3)
        self.btn_ok.grid(column=1, row=3)

        for child in self.winfo_children():
            child.grid_configure(padx=2, pady=2)

    def save_settings(self):
        self.model.uman.start_group()
        self.model.ed.set_property(self.model.song_config, "name", self.var_song_n
ame.get())
        self.model.change_pattern_length(self.var_pattern_length.get())
        self.model.change_sequence_length(self.var_sequence_length.get())
        self.model.uman.end_group()
        self.destroy()
```

---

tick_manager.py

```
from model import Model

class TickManager:
    """An advanced counter for position in a song."""

    def __init__(self, model: Model, ignore_loop: bool = False):
        self.model = model
        self.ignore_loop = ignore_loop
        self.mono_tick: int = 0
        self.sequence_enabled: bool = False
        self.bar_number: int = 0
        self.pat_tick: int = 0

    def next_tick(self):
        result = (
            self.mono_tick,
            self.sequence_enabled,
            self.bar_number,
            self.pat_tick
        )
        self._increment_tick()
        return result
```

```
    def set_tick(self, bar_number: int, pat_tick: int):
        self.bar_number = bar_number
        self.pat_tick = pat_tick

    def enable_sequence(self):
        self.sequence_enabled = True
        self.model.event_bus.bar_playing(self.bar_number)

    def disable_sequence(self):
        self.sequence_enabled = False
        self.model.event_bus.bar_playing(-1) # -1 means no bar is playing

    def _increment_tick(self):
        # increment tick
        self.mono_tick += 1
        if self.sequence_enabled: # normal pattern playback
            self.pat_tick += 1
            self._justify_tick()

    def _justify_tick(self):
        if self.pat_tick >= self.model.song_config.get_property("pattern_length"):
            self.pat_tick = 0
            self.bar_number += 1

            # respect loop markers
            if ((self.bar_number == self.model.song_config.get_property("loop_end"
))
                    and self.model.song_config.get_property("loop_enabled")
                    and not self.ignore_loop):
                self.bar_number = self.model.song_config.get_property("loop_start"
)

            # stop playing through song if we reached the end
            if self.bar_number >= self.model.song_config.get_property("sequence_le
ngth"):
                self.disable_sequence()
            else:
                self.model.event_bus.bar_playing(self.bar_number)
```

---

top_frame.py

```
import tkinter as tk
import tkinter.ttk as ttk
from tkinter import filedialog as fd

from model import Model
from song_settings import SongSettings
from playback import Playback
from audio_exporter import AudioExporter

class TopFrame(ttk.Frame):
    """
    UI component - holds the top bar of buttons.
    Includes playback control, undo/redo and song init/load/save/export/settings."
""
```

```
    def __init__(self, parent, *args, model: Model, playback: Playback, audio_exporter: AudioExporter, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.parent: tk.Tk = parent
        self.model = model
        self.playback = playback
        self.audio_exporter = audio_exporter

        self.btn_play = ttk.Button(
            self, text="⍰", width=3,
            command=self.playback.play
        )
        self.btn_pause = ttk.Button(
            self, text="⍰", width=3,
            command=self.playback.pause
        )
        self.btn_stop = ttk.Button(
            self, text="⍰", width=3,
            command=self.playback.stop
        )
        self.btn_undo = ttk.Button(
            self, text="↶", width=3,
            command=self.model.uman.undo
        )
        self.btn_redo = ttk.Button(
            self, text="↺", width=3,
            command=self.model.uman.redo
        )

        def new_song_callback():
            self.model.init_tree()
            self.parent.title(f"project noteblock - new song")

        self.btn_new_song = ttk.Button(
            self, text="☆", width=3,
            command=new_song_callback
        )

        def load_callback():
            filename = fd.askopenfilename(
                defaultextension=".json",
                filetypes=[("JSON project file", "*.json")],
                title="Load song project file..."
            )
            self.model.from_file(filename)
            self.parent.title(f"project noteblock - {filename}")

        self.btn_load = ttk.Button(
            self, text="🗀", width=3,
            command=load_callback
        )

        def save_callback():
            filename = fd.asksaveasfilename(
                defaultextension=".json",
                filetypes=[("JSON project file", "*.json")],
                title="Save song project file..."
            )
```

```
            self.model.to_file(filename)
            self.parent.title(f"project noteblock - {filename}")

        self.btn_save = ttk.Button(
            self, text="💾", width=3,
            command=save_callback
        )

        self.btn_export_wav = ttk.Button(
            self, text="↑", width=3,
            command=lambda: self.audio_exporter.export(fd.asksaveasfilename(
                defaultextension=".wav",
                filetypes=[("WAV audio file", "*.wav")],
                title="Export song as WAV..."
            ))
        )
        self.btn_settings = ttk.Button(
            self, text="⚙", width=3,
            command=lambda: SongSettings(self, model=self.model)
        )

        self.btn_play.grid(column=0, row=0)
        self.btn_pause.grid(column=1, row=0)
        self.btn_stop.grid(column=2, row=0)
        self.btn_undo.grid(column=3, row=0)
        self.btn_redo.grid(column=4, row=0)
        self.btn_new_song.grid(column=5, row=0)
        self.btn_load.grid(column=6, row=0)
        self.btn_save.grid(column=7, row=0)
        self.btn_export_wav.grid(column=8, row=0)
        self.btn_settings.grid(column=9, row=0)
```

---

undo_manager.py

```
from collections import deque
from node_actions import Action

# NOTE: group depth keeps track of how many layers deep we are supposed to be in g
roups
# so that if you start_group twice you have to end_group twice to actually finish
the group
# this allows us to pretend to have groups within groups

class UndoManager:
    """Manages a timeline of actions that can be traversed."""

    def __init__(self, past_len: int = 10, future_len: int = 10):
        self.past: deque[Action | list[Action]] = deque(maxlen=past_len)
        self.future: deque[Action | list[Action]] = deque(maxlen=future_len)
        self.current_group: list[Action] = []
        self.group_depth: int = 0

    def reset(self):
        self.past.clear()
        self.future.clear()
        self.current_group = []
        self.group_depth = 0
```

```python
    def perform_without_undo(self, new_action: Action):
        new_action.perform()

    def perform(self, new_action: Action):
        new_action.perform()
        if self.group_depth > 0:
            self.current_group.append(new_action)
        else:
            self.past.append(new_action)
        self.future.clear()

    def start_group(self):
        self.group_depth += 1

    def end_group(self):
        if self.group_depth > 0:
            self.group_depth -= 1
            if self.group_depth == 0:
                self.past.append(self.current_group)
                self.current_group = []

    def can_undo(self):
        return len(self.past) > 0

    def undo(self):
        if self.can_undo() and self.group_depth == 0:
            action = self.past.pop()
            if isinstance(action, list):
                for act in reversed(action): act.undo()
            else:
                action.undo()
            self.future.append(action)

    def can_redo(self):
        return len(self.future) > 0

    def redo(self):
        if self.can_redo() and self.group_depth == 0:
            action = self.future.pop()
            if isinstance(action, list):
                for act in action: act.perform()
            else:
                action.perform()
            self.past.append(action)
```

PROJECT TESTING

Testing strategy

The main goal I had when recording the testing video was to demonstrate a typical music production workflow in the program, while still providing a reasonably comprehensive showcase of the program's features, as listed in the objectives in the project analysis.

Test table

Link to video: https://www.youtube.com/watch?v=YcFQi5cvCZo

Objective numbers as listed in the analysis document are referenced where relevant. Note that video timestamps are not comprehensive – some actions are performed many more times during the testing video than are listed.

Test no.	Obj	Description	Video	Comments	Pass /fail
1	2a, 2b	Changing a channel's instrument	0:42, 2:06	Note block instruments can be selected via a drop-down menu in the instrument editor and are played/exported correctly.	Pass
2	2c, 3b	Sustained notes	2:51	Sustained notes function as expected and their instrument can be changed separately to the main instrument.	Pass
3	3a	Adding notes in piano roll	0:11, 1:13	Clicking in the piano roll adds notes to the selected pattern.	Pass
4	3d	Pattern stretching	7:50	Patterns are stretched correctly, affecting the speed of playback.	Pass
5	4a, 4d	Adding pattern to sequencer	0:19, 1:10	Patterns can be dragged from the pattern list into the sequencer.	Pass
6	4b	Channel mute	4:35	Muted channels are not played.	Pass
7	4b	Channel solo	1:47, 2:49	If any channels are soloed, only those channels play, and the rest are muted.	Pass
8	4b	Changing channel volume	3:01	This allows the user to mix channels to achieve a better balance of sounds.	Pass
9	4c	Looping playback	0:32, 4:42	If looping is enabled, playback continues indefinitely within the loop markers.	Pass
10	5	Adding effects	3:08	A delay effect is added and tweaked to create an echoing effect.	Pass
11	5a, 5f	Applying effects in series	3:29	A second delay effect is added in series, and together they create a stereo delay.	Pass
12	7a	Smooth, low-latency playback	0:33, 4:42	There are no discernible glitches in playback, even over extended periods of time such as in the live performance.	Pass
13	7c	Live edits affecting playback	0:42, 1:13	Changing things such as channel instrument and notes in a pattern affect playback in real time with low latency.	Pass
14	8a	Saving a song	4:02	The test song is saved to the custom JSON-based format.	Pass
15	8a	Loading a song	4:13	The test song is loaded back in.	Pass

16	8b	Exporting audio of a song	7:20	The test song is exported as a WAV file, which sounds identical to playback within the program.	Pass
17		Resizing window	0:04, 4:33	The window can be resized freely, and UI components arrange themselves efficiently within the window.	Pass
18		Creating patterns	1:00, 1:48	Patterns can be created in the pattern list.	Pass
19		Creating channels	1:39	Channels can be created in the sequencer.	Pass
20		Changing pattern and channel colours	1:06, 1:43	This helps the user to navigate around a song more easily.	Pass
21		Undoing and redoing edits	1:25, 2:27	Edits are undone and redone without any problems.	Pass
22		Live performance	4:35	Playback looping and channel muting are used extensively in a live performance of a demo song.	Pass

PROJECT EVALUATION

Review of Objectives

Aspirational objectives have been omitted except where achieved

1. **General interface**
    a. UI layout: achieved
    All the UI components originally listed in the design have been implemented in the program, except for the BPM and time signature settings, which I soon realised was not very compatible with Minecraft's music mechanics, and which I have effectively replaced with the single "pattern length" setting in the song configuration dialog.
    b. Clean, modern interface: partially achieved
    The interface conforms to the UI styling of the platform it is run on and appears relatively modern. Unfortunately, I was unable to implement hi-DPI support to a satisfactory degree, primarily due to the limitations of the GUI library, tkinter.

2. **Instruments**
    a. Access to all default Minecraft note block sounds: achieved
    All note block instruments from the current version of Minecraft (1.19) can be selected in the instrument editor, and their audio is played/exported correctly.
    b. Select a single instrument per channel: achieved
    c. Select a main and sustain instrument per channel: achieved
    Both the main and sustain instruments can be chosen in the instrument editor, and they function correctly, with the sustain instrument playing notes repeatedly to simulate the effect of a held note.

3. **MIDI-like pattern editor**
    a. Simple scrollable and editable piano roll: achieved
    The piano roll is interactive and can be zoomed in and out as well as scrolled. Editing a pattern accomplished by left-clicking to place a note and right-clicking to delete it.
    b. Per-note sustain setting: partially achieved
    Notes in a pattern do not have intrinsic sustain settings. I ended up implementing sustained notes in a different way, by specifying a special "sustain off" note and instructing channel instruments to sustain notes until they receive such a note.
    c. Play notes as they are selected/edited: failed
    Unfortunately, I was unable to implement this feature, as it would require a significant reworking of the playback system. In the current system, patterns do not have their own instruments, but are played using the instruments of any channel which has that pattern in its sequence.
    d. *Arbitrary pattern lengths: partially achieved*
    Patterns do not have individually specified pattern lengths, but the length of all patterns can be changed simultaneously in the song configuration dialog, allowing for patterns of any length.

4. **Sequencer**
    a. Simple scrollable grid-constrained sequencer: achieved

The sequencer can be scrolled both vertically (across channels) and horizontally (over time), and patterns can be placed in the intersections of channels and bars, following a grid-constrained system.

b. Channel headers: achieved
   Each channel has a header, which is always visible to the right of the sequencer even as it is scrolled horizontally. Channel headers have volume and pan settings, and mute and solo buttons, all of which are fully functional. Clicking a channel opens its instrument and effects in the instrument editor and effect rack at the bottom of the interface.
c. Playback loop markers: achieved
   Loop markers for playback only (not for audio exports) are visible along with bar numbers above the main sequencer display. The loop can be set by right-click dragging across the desired bars and can be enabled or disabled with a button in the sequencer.
d. Drag and drop pattern placement: achieved
   Patterns can be dragged from the pattern list on the left into the sequencer and are added to the correct channel and bar.

5. **Effects**
   a. 3 effect slots per channel: achieved
      Each channel can have 3, or a lot more than 3, effects, applied in series.
   b. Drop-down effect menus to add effects to slots: achieved
      The drop-down menu to the right of the effect rack allows any effect type to be added to the end of the effect rack, from where it can then be moved into the desired position in the effect chain.
   c. Ability to swap effects between slots/delete effects from slots: achieved
      Effects can be reordered or deleted using the buttons in the header of each effect.
   d. UI sliders and knobs for effect control: partially achieved
      Effect parameters can be controlled via direct text input or pressing up/down buttons, however I was unable to implement sliders and knobs.
   e. Visualizers for selected effects: failed
      Unfortunately, real-time effect visualizers were something I was unable to implement, mostly due to the limitations of the GUI library, tkinter, which is slow enough already even without having to handle rendering live visualizers.
   f. *Unlimited, scrollable effect slots per channel: achieved*
      A channel can have any number of effects, and there is a scroll bar to scroll through the effect rack if there are too many effects to fit on screen.

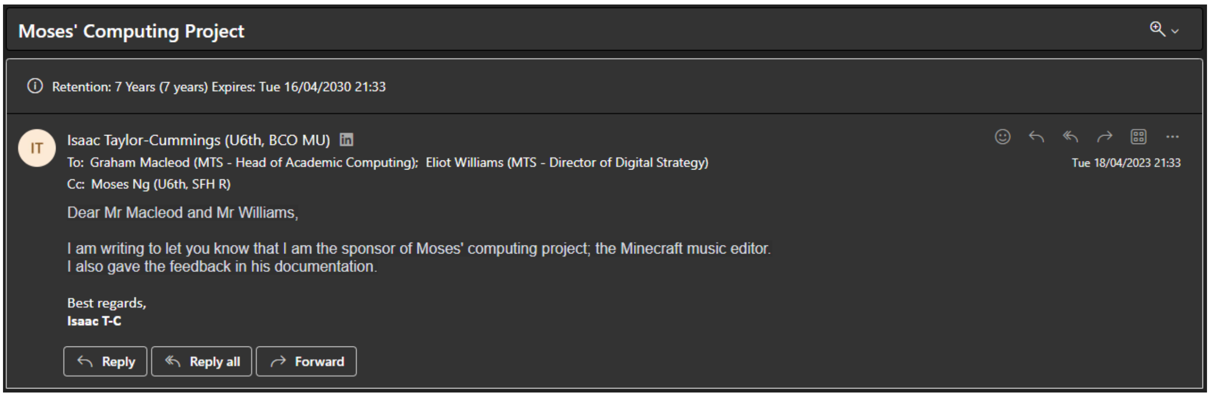
7. **Playback**
   a. Smooth, low-latency song playback: achieved
      Song playback is real-time and free of glitches, with everything playing correctly. It can be started and stopped anywhere in the song using the user interface, with less than one second of delay between user action and playback response.
   b. Per-pattern playback: partially achieved
      Playing patterns individually has not been directly implemented but can easily be accomplished by placing a pattern on a soloed channel.
   c. *Ability for live edits to affect playback: achieved*

Edits to a song affect playback responsively in real time, with less than one second of delay between making an edit and being able to hear its effect. This works even while the song is playing.

8. **File I/O**
   a. Custom file format for saving and loading songs: achieved
   Songs can be saved to and loaded from a JSON based format, which stores all the information about the nodes comprising the song data that is required to fully reproduce the song.
   b. Ability to export to WAV audio files: achieved
   The audio export dialog allows the user to export WAV audio files, using the same audio generation algorithms used in live playback.
   c. Per-channel audio export: achieved
   This can be accomplished by simply soloing the desired channel so only it can be heard, then exporting audio.

9. **In-game implementation: failed**
   I underestimated the complexity of exporting songs into Minecraft itself and was ultimately unable to complete this part of my project (described in more detail in the improvements section). However, the program still functions effectively as a tool for creating Minecraft-style music, and the modular nature of my code means that integrating a new exporting algorithm into the project in the future would not be difficult.

Sponsor feedback



Moses' Computing Project

Retention: 7 Years (7 years) Expires: Tue 16/04/2030 21:33

Isaac Taylor-Cummings (U6th, BCO MU)
To: Graham Macleod (MTS - Head of Academic Computing); Eliot Williams (MTS - Director of Digital Strategy)
Cc: Moses Ng (U6th, SFH R)
Tue 18/04/2023 21:33

Dear Mr Macleod and Mr Williams,

I am writing to let you know that I am the sponsor of Moses' computing project; the Minecraft music editor.
I also gave the feedback in his documentation.

Best regards,
Isaac T-C

Reply | Reply all | Forward

Me: What do you think of the program's user interface?

Isaac: I appreciate how user friendly the user interface is. Everything is so simple to understand – looping is easy, mixing is easy, putting in notes is easy, and you can do things like change the tempo with a few simple clicks. However, it looks rather dull and grey – to me, it looks more like a Windows pop-up than a DAW. I would like to be able to customise the theme of the user interface; even just the ability to switch up the colours could make a big difference.

Me: How about the workflow of the program? Did you find it efficient to use?

Isaac: It is more efficient than making music in Minecraft, for sure! Things like being able to repeat single melodies or entire sections of a song with ease, changing instruments on the fly, adding space to the sound with effects, and having a piano roll to input notes are a significant improvement over doing those things in game. In addition, being able to undo changes so easily means that I can quickly try out different ideas in a song without worrying about losing progress. However, I found that having to use the mouse for everything is rather finicky, and I would appreciate the addition of keyboard shortcuts.

Me: What did you think of the audio side of things – playback and exporting audio?

Isaac: Well, I have experienced no audio glitches, and the sound is consistent between playback and exporting, as well as with the actual game. Not much else to say, except that it works perfectly as far as I can tell.

Me: Are there any features you would want to be added in particular?

Isaac: I would like it if some parts of the user interface were more like GarageBand. For example, having faders for mixing and knobs for effects on screen would be more intuitive for me. It would also help me work faster if I could select multiple pattern placements in the sequencer and deal with them together, for example moving all of them around or duplicating them to quickly extend a song. And finally, the feature I would like most is being able to export songs to Minecraft! If you ever get that made, do please let me know.

Me: Overall, how satisfied are you with the program?

Isaac: I am very happy with it. I will find it useful for making Minecraft-style music, with or without the exporting of schematics, and I may in fact use it to draft songs to then build by hand in game – it will help me find the right instrumentation for a song and write melodies and harmonies much more quickly.

Analysis of sponsor feedback

It is worth noting that my client is only able to evaluate the end user experience: the visuals, workflow and user-exposed functionality of the program. Hence, the details of the technical implementation underlying much of the program could not be discussed in the interview.

My client largely expressed satisfaction with the visuals and workflow of the program, especially by the efficiency of creating music in the program when compared to doing so manually in Minecraft itself. However, he has still suggested several features that he would additionally want, including an improved appearance, keyboard accessibility, and interface components that he is more familiar with.

Possible improvements

Based on my sponsor's feedback, as well as my own views based on the experience of putting the project together and testing it, I have come up with the following list of improvements that I would make to the project, given more time:

1. The biggest improvement, and the most time-consuming, would be rewriting the project in a different language, using a different GUI framework. My project in its current state is limited somewhat by the speed of Python, and more significantly by tkinter, the GUI framework that I chose, due to its slow speed, limited capabilities (especially regarding scrolling and drawing of arbitrary shapes), and the tedious nature of designing UI layouts with it. In hindsight, using HTML/CSS/JS, with the addition of frameworks such as React for responsive components and Bootstrap for clean UI theming, would likely have been a better choice.

2. I would add a way to theme the user interface. In addition to simply changing the colours of the interface, the shapes of components such as buttons and text fields could be customised. Being able to change the theme and see its effect while the program is running would be ideal; this would probably require integrating the theme system with the event system, so that UI components could respond to the theme being changed.

3. I would add user interface components such as sliders and dials to edit numeric values such as volume and pan. Dials would respond to dragging the mouse vertically, and in this way they would behave similarly to sliders, just with a smaller footprint on screen. I could also let the sliders and dials still accept keyboard input, to allow the user to input precise values if desired.

4. I would implement keyboard shortcuts to perform common actions in the user interface more easily, for example pressing the spacebar to play or pause playback. The keyboard shortcut system would have to be tied to the program as a whole, as it needs to accept keyboard input across the whole program and be able to trigger any action in the program.

5. I would implement mechanisms for selecting groups of notes in a pattern or placements in the sequencer and moving, deleting, copying and pasting them, in order to make the workflow of the program more efficient.
   In the piano roll for example, when the user drags the mouse to create a selection, the bounding box of the drag input could be calculated, and any notes within that bounding box would be selected. Adding hotkey functionality to these mechanisms that can modify their behaviour would also be helpful, for example shift-clicking in the piano roll to allow selecting or deselecting individual notes, and this would likely have to be handled partly by the keyboard shortcut system.
   As for copying and pasting, due to the custom data structures in the program, I could either implement a clipboard system which operates only within the program, or hook into the system clipboard by saving and loading the JSON representation of the objects. In either case, I would have to consider that many song objects have some unique properties, such as an ID, and this would have to be changed appropriately when copying and pasting.

6. The biggest objective that I ultimately failed to accomplish was exporting songs into Minecraft. Among other things, this would involve:
   - Designing an in-game layout. It must be tileable, as the length of a song determines the length of the in-game structure. It must also be flexible in the relative positions of different redstone circuits in order to adapt to different volume and pan settings, and this additional complexity means that the layout would have to be an algorithm for generating circuit positions instead of a single layout.
   - Writing a library for encoding to and decoding from the binary schematic file format and translating the previously designed in-game layout algorithm into code.
   - Designing a way to deal with the problem of multiple notes in the same tick having the same volume and pan. This would technically mean that they need to occupy the same coordinates in the schematic, which is not possible; I would have to investigate the best compromise in terms of block positioning.
   - Coming up with a way to deal with the huge sizes of the resulting in-game layouts. In my current best concept for a possible circuit layout, one second of music would translate to at least 10 blocks in length of the circuit, so a three-minute song would

be 1800 blocks in length. The circuit would also have to be up to 100 blocks in width and height in order to accommodate the possible volume and pan settings of individual notes. Combining these dimensions together and you get a very large schematic that the game would struggle to import, if it could even do so at all.

All of this would be a significant undertaking of its own, and hence I was unable to complete it along with the rest of the project. However, it is something that I may try to tackle in the future, after the submission of the project.

7. In addition to exporting Minecraft schematics, I would also add support for exporting other formats, such as MIDI files from patterns or channels, and MP3 files for listening to a song without the program or a large WAV file.