Bilkent University

Department of Computer Engineering

Senior Design Project

microgliss: a microtonal editing tool for the world music

Final Report

Group Members: Artun Cura, Sonat Uzun, Orkan Öztrak

Supervisor: Vis. Prof. Dr. Fazlı Can

Innovation Expert: Burcu Coşkun Şengül

Final Report
April 30, 2021

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS492

Contents

1. Introduction

Microgliss is a note editor and a virtual instrument that allows users to write in polyphonic glissandos, flexible dynamics and microtonal notes using a node and edge-based workflow. It allows users to write music in any tuning system or independent from any tuning system. It allows users to add lines between notes, with editable curves and dynamics which controls velocity and frequency interpolation. Its synthesizer has different options that lets users design very intimate and colorful sounds which can be used for rich compositions. Microgliss is designed by keeping in mind common limitations of existing midi/note editors and synthesizers and growing interest in microtonal music around the globe.

This project started as a tool, but it has become a digital instrument which has its own unique sound and capabilities. As developers, we can't help but feel as luthiers which designed an amazing instrument. We are proud of how it started, how it developed and where it is right now.

This digital instrument is developed for use of musicians. To understand the details about it, you need to have basic knowledge about the terms which will be used during the discussion of the project. These terms are explained in the next section.

Definitions, Acronyms and Abbreviations

Note: A symbol denoting a musical sound.

Nodge: A term we invented (and used frequently in the code.) It means both node and edge. Eg: "Ctrl+A lets user select all nodges."

Semitone: The 12 interval that each correspond to a note in the Western 12-tone tradition.

Octave: The entirety of the 12 semitones. An octave is also the distance between two same lettered notes.

Microtonal Music: Music that consists of intervals that are smaller than a semitone. This term also includes any tuning system which differs from the western 12-tone tradition.

Synthesizer: An musical instrument that generates audio signals. Original synthesizers are analog and contain circuits. As the technology evolved, their digital versions developed as well.

Glissando: Sliding from one note to another seamlessly.

DAW: Digital Audio Workstation. Comprehensive musical programs that are designed for recording, editing, and producing music.

Automation: Automation in the context of DAW's is lines usually drawn by hand to change parameters of a track (eg: volume/pan) and/or a plugin (eg: mix) over time.

VST: A plug-in format for a digital audio workstation.

OSC: Open Sound Control. A protocol for networking sound synthesizers, computers, and other multimedia devices for purposes such as musical performance.

MIDI: Musical Instrument Digital Interface. A communication protocol for musical instruments and controller devices.

Preset: Pre-saved synthesizer settings. They can also be produced by different professional sound-designers for end-user use.

Sample: A sample, which is a floating point value, is the smallest component of the digital representation of the sound. A one second sound stored in a computer consists of over 40000 samples (usually 44100 or 48000). These samples are written into buffers and speakers vibrate according to the data in the buffers, therefore generating sound. Each sample represents subtle changes in air pressure which we perceive as sound.

ADSR Envelope: To control how sound changes over time, Attack, Decay Sustain and Release parameters are used.

2.0 Requirements Details

2.1 Functional Requirements

2.1.1 General

- Users can import *.scl files, and change grid properties of the editor.
- Users can load, and save compositions in *.txt format.
- Users can load midi files.
- Users can change the output volume using a master volume controller.
- Users can change the size and scale of Microgliss, UI will react.
- Microgliss saves every data edited by the user (nodes, edges, editor sizes, zoom, edited parameters etc.).

2.1.2 Editor

- The users can add nodes to the editor.
- Users can specify their rhythmic, frequency and velocity properties of nodes.
- Users can select nodes, color of the selected nodes changes to indicate selection.
- Users can deselect nodes.
- Users can connect selected nodes to another node with edges.
- One node can connect to many other nodes.
- Many nodes can connect to one node.
- Users can edit and remove nodes.
- Edges react to the change of the nodes.
- Users can zoom in/out of the editor.
- Users can select multiple nodes at once.
- Editor gives audio feedback with respect to frequency of the node, whenever a node is edited.
- Users can turn on/off audio feedback.
- Users should be able to specify rhythm (a division of time).
- Users can turn the snap for rhythm and frequency grid on and off easily.
- Users can change the velocity value (how loud they are) of nodes, node color (alpha) changes respectively.
- Users can change velocity of multiple nodes at the same time.
- Users can change the curvature of the edges.
- Users can copy and paste nodges.
- Users can drag nodges.
- Visible portion of the grid moves towards currently dragged nodges if the cursor is near the edges of the window.
- Users can see the beat data from the horizontal part of the editor.
- Users can see the frequency data from the vertical part of the editor.

2.1.3 Synthesizer

In this section, everything addressed as sound represents the sound created with respect to node and edge data of the Editor.

- Users can modify the ADSR envelope of the sound.
- Users can use a filter that changes the frequency data of the sound.
- Users can modify the ADSR envelope of the filter, and the cutoff.
- Users can choose different waveforms.
- Users can mix different waveforms.
- Users can apply ring modulation (multiplying them while mixing instead of adding) to the waveforms.
- Users can change the unison count (voices that are playing the same note) and detune them (change their frequency slightly) to create interesting sounds and phasing effects (quality of the sound changes in long intervals of time).

- Users should be able to load pre-saved synthesizer settings (load presets).
- Users can edit these parameters, also they can be edited by the host(DAW). This lets users automate parameters directly from the DAW.(Automation in the context of DAW's lines usually drawn by hand to change parameters of a track (eg: volume/pan))

2.2 Nonfunctional Requirements

Compatibility: Microgliss should run on all DAW's which support VST format. Users should be able to import MIDI files generated from different applications or read the data directly from DAW.

Responsiveness: Microgliss should be able to communicate the changes made in it to the synthesizer,real-time and during playback. This aids in the workflows of the users.

Ease of use: Microgliss should be easy to use, due to its main focus being musicians which might have different kinds of technical knowledge.

Robustness: Microgliss should crash as little as possible as crashing may mean the loss of work of the users.

Performance: Microgliss should at least be able to playback sounds in real time. Performance is really important because the better it gets, users will be able to use it in conjunction with different plugins as well as other instances of Microgliss.

3. Final Architecture and Design Details

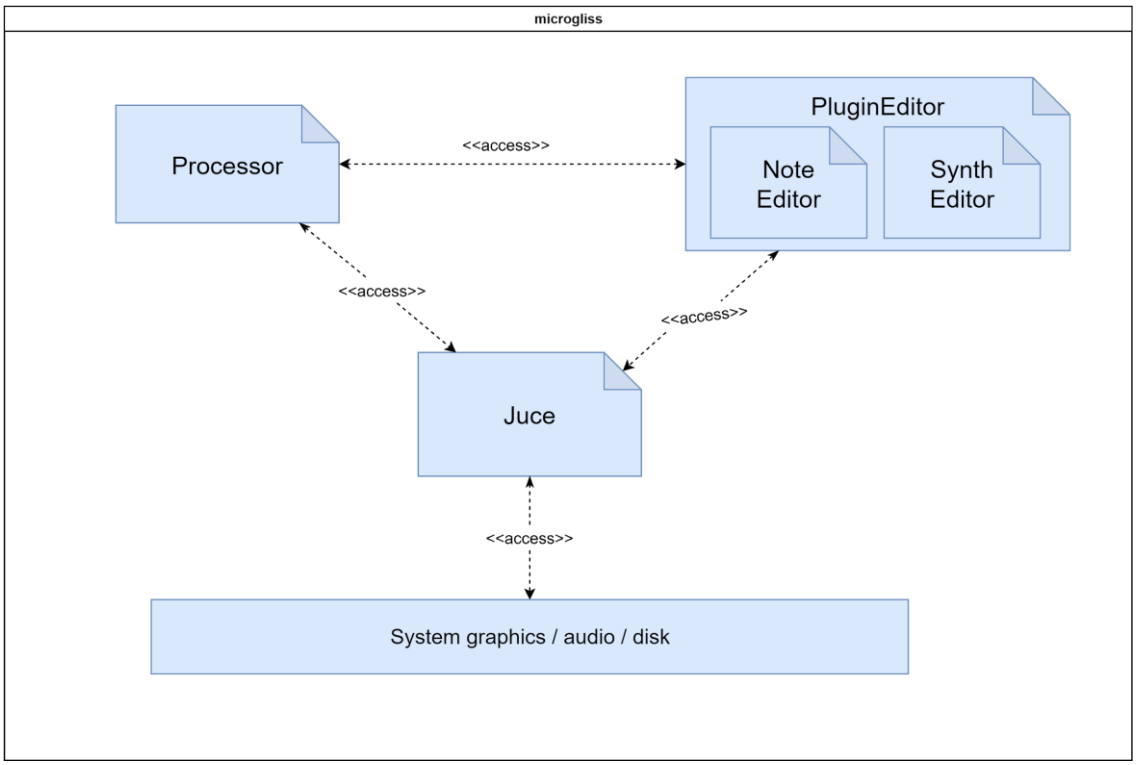
As it is stated in the previous reports, during the development of the project, we used C++ and Juce framework. Microgliss might be used as a plugin, so it should be wrapped in different file formats to run appropriately in different digital audio workstations. This affects how the architecture is shaped. For this report, we assume that the purpose of this section is showing the overall structure and what we've done, instead of making a class diagram that is implementable right away just by looking at it. Due to that, we will omit unnecessary details of JUCE, and only discuss the details we wanted to discuss that contribute to overall structure.

One of the things that we've changed during the final implementation is OSC connection. OSC(Open Sound Control) is a protocol for controlling synthesizers on a network, and we were thinking that providing this kind of connection from our editor to other synthesizers would be useful, because we were not planning to focus on our synthesizer and Microgliss would just be a tool to control existing synthesizers. But during the implementation process we've seen that we are capable of implementing

a functional synthesizer, and using OSC protocol wouldn't be that useful for us. That's why all details about the OSC protocol are removed in this report.

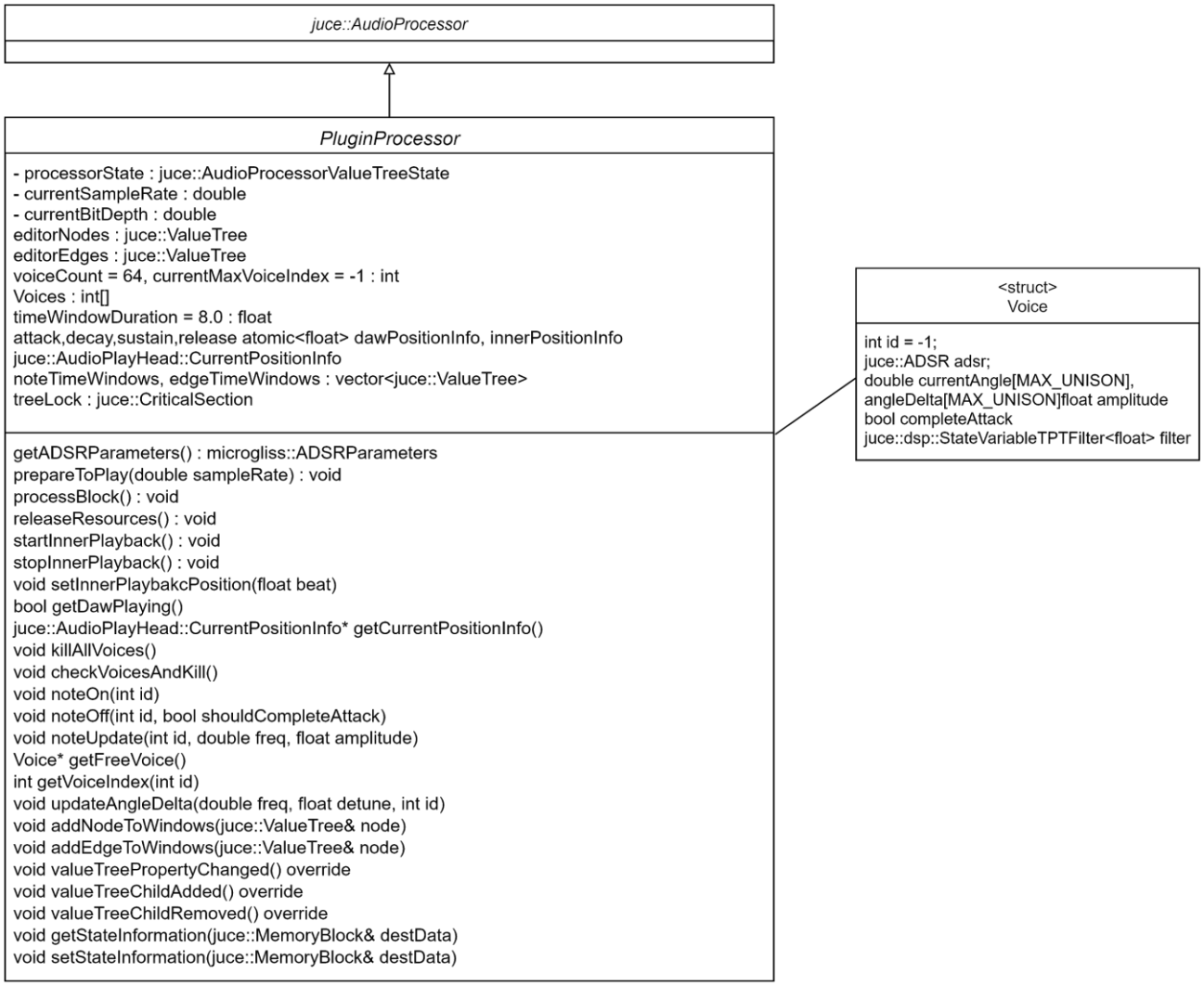
3.1 Basic class diagram

From our standpoint, the structure of the Microgliss is as shown:



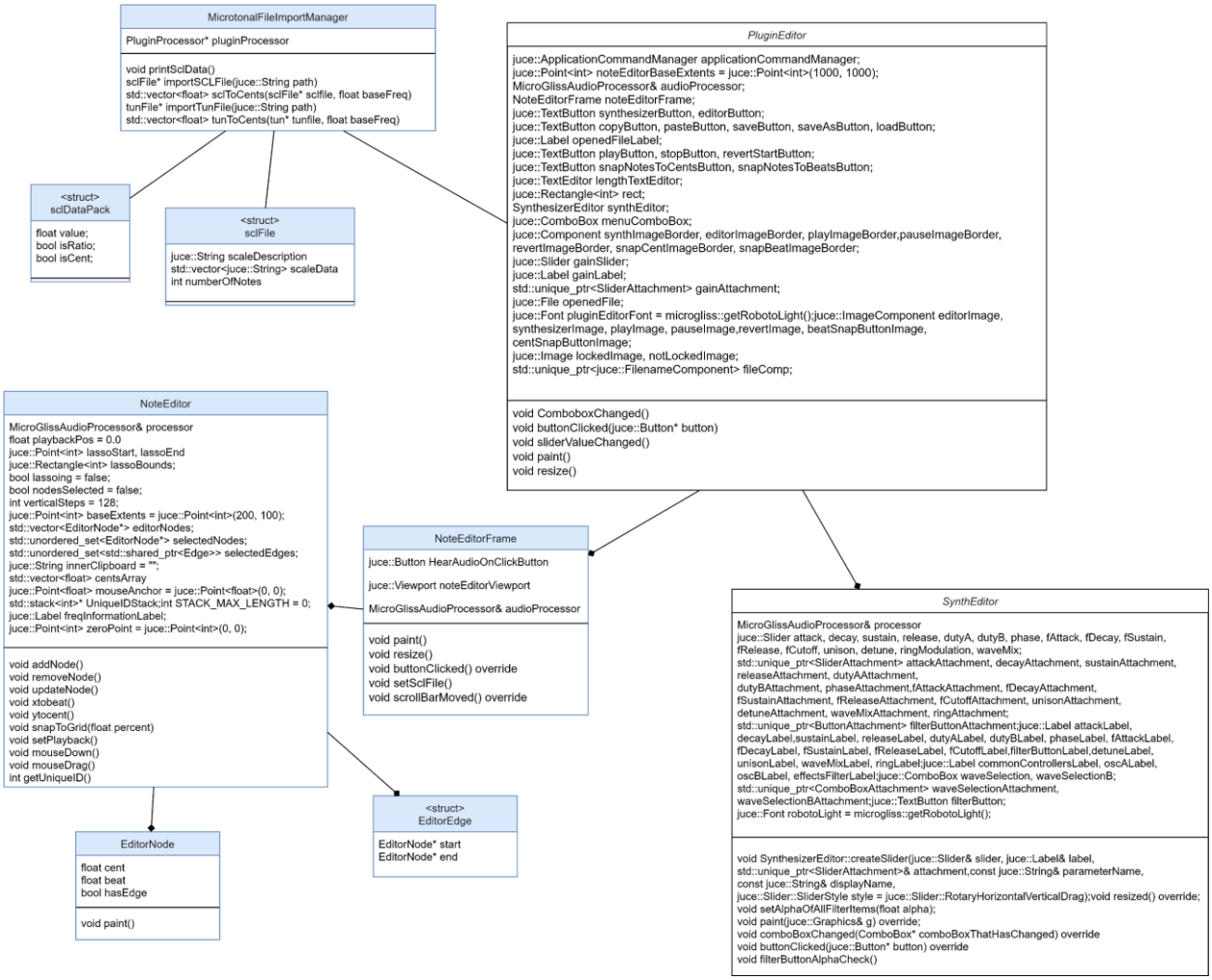
From the diagram, it can be seen that the layers we've written are standing at the same level and performing operations with the Juce framework. Juce provides basic buttons, sliders, file load etc. for the editors, and provides basic data structures, audio buffers, DAW controls for the data side of the editors and processor. Most of these functions are overridden for improvement, but their basic purpose is the same. Processor part is the sound generating part of the Microgliss, while others provide data structures and UI based functions for the user to use. All changes done in synthesizer and note editor layers affect how the final sound is shaped in the processor part. While class diagrams of the note editor and plugin editor will be discussed next, details of their implementation/development will be discussed in Section 4.

3.1.2 Processor



The processor part is the heart of Microgliss, and it is composed of a single class which performs all DAW and Standalone processing operations. Starting, ending or updating note operations etc. are in this class. Voices are the simultaneous notes that are playing at the same time and structs are used to save this data in a compact form. AudioProcessor contains a Voices array that is initialized with 64 voices at the start. After Microgliss starts playing notes, properties of these voices change.

3.1.2 Plugin Editor



Plugin Editor is composed of two main parts, which are Synth Editor and Note Editor. Instead of using Note Editor directly, it uses the Note Editor's wrapper, which prints a frame in the program, that shows the details like note names and beat information.

3.2 Hardware/software mapping

Microgliss runs on any DAW that supports the VST3 format. Users can import MIDI files generated from different applications. Microgliss also can run standalone in Windows. Microgliss doesn't have any hardware mapping other than the mappings of the DAW or Windows.

3.3 Global Software Control

General:

- The program uses threads.
- Many parts of the code reads and writes node and edge data. Locks are used often to ensure stability.
- Synthesizer parameters are hard linked to the processor and DAW, using thread-safe solutions implemented in JUCE.

Processor:

- Holds nodge info.
- Holds any other information about the composition, synthesizer and editor that needs to persist.
- Saves and loads the data when requested by the DAW.
- Optimizes the nodge data whenever a change has been made to it.
- Plays the composition when requested by user or DAW (user requests the playback through DAW). DAW requests have priority.
- Gives audio feedback to the user while editing (if the necessary option is turned on).
- Apply locks whenever reading or writing nodge data in order to preserve stability.
- Automatically react to changes in synth parameters.

Note Editor:

- Reads and reflects the data in the processor on startup.
- Modify the data in the processor.
- Use locks when modifying nodges to ensure stability.

Synthesizer Editor:

- Modify synthesizer parameters.
- Reflect the changes in synthesizer parameters.

3.3 Persistent data management

Different kinds of data persists in the system: parameters of the synthesizers and compositions made in the note editor, the state of the editor (eg: zoom level). When using the program as a plugin, DAW triggers the save and load functions. But inside of the save and load functions have to be explicitly written by us. We used JUCE's Value Tree data which easily converts the data it contains into XML format. Then we can convert this data to binary and save it into memory and/or disk.

In addition to being able to save and load the entire state of the plugin within DAW, compositions are saveable/loadable in txt form. Composition data excludes synthesizer parameters and editor state.

3.4 Boundary conditions

3.4.1 Plugin set-Up

To set-up Microgliss, the user (manually) needs to copy the VST3 file to their currently existing VST Plugins folder. These folders are the locations where the DAW will look for plugins, they are predefined by the user. After the folders are rescanned the plugin can be imported to the DAW and start running.

3.4.2 Standalone Set Up

Standalone version can be run directly from the exe without any additional set-up.

3.4.3 Plugin Initialization

Processor part of the editor can be initialized by the DAW without initializing the editor for optimization. Plugin host has the control of which parts of the plugin will be initialized.

3.4.4 Editor Initialization

In the standalone version, Editor is initialized whenever the program is started. In the plugin version, it is initialized whenever it is requested by the DAW. In initialization the editor reads and reflects the data in the processor.

3.4.5 Editor Deletion

In the standalone version it is deleted whenever the program is closed. In the plugin version, the editor is usually deleted whenever it is not visible in the DAW, for optimization purposes.

3.4.6 Host Save

Host requests the plugin to save itself whenever it is saving its own state.

3.4.7 Host Deletes A Plugin Instance

Host also requests the plugin instance to save itself while deleting the instance so that it’s state can be recovered with an undo operation.

3.4.8 Program Termination

We used smart pointers and JUCE’s classes that handle their own memory to prevent memory leaks.. All data is cleared when the program is fully terminated.

3.0 Development/Implementation Details

3. 1 Programming Languages/ Framework & Tools

During the implementation of Microgliss, we used C++ programming language, and JUCE, a framework which is specialized for GUI and Audio application development. For the task management we used Trello. In the rest of this section details we want to share will be discussed.

3.1.1 C++

3.1.1.1 Memory Management:

We were trusting our memory management knowledge from previous Bilkent courses, so during the first steps of the projects we used basic pointers and tried to handle memory management by ourselves, by using the new keyword and “*” operator directly.

Later we realized that under the name of modern C++ JUCE some classes such as the ValueTree handles memory and references the same way a Java class would: without explicitly declaring new pointers and using the “*” operator. After realizing this we have refactored our code to take full advantage of the simplicity provided by this implementation.This way of referencing variables wasn’t thread-safe so we had to include many locks in our code where ValueTree data is read and changed. But this would probably be the case if we used new and * explicitly as well.

We also switched to using smart pointers for our own data structures in the later phases of the project. Smart pointers made memory management easier and safer for us.

3.1.1.2 Dynamic casting:

Dynamic casting is another concept that we learned along the way and became very useful in the project. Basically it is trying to cast an object to another, and if it is not castable, it ends up as a null pointer. This way it also allows us to check the types of an object in C++. An example use of it was for detecting whether the click performed on a node. We were getting the component clicked, it returns a Component object and tries to cast it to an Editor Node. If it is null, that means we didn’t click on a node; if it is not we click on a node so perform necessary operations.

3.1.2 JUCE

As developers, we think that even though JUCE helped us a lot through the way, it also had some interesting behaviours that we find hard to understand. Sometimes we are amazed by the things that are implemented in JUCE, but sometimes we are surprised how they didn't implement the most basic must features.

3.1.2.1 Automations

Automations in the context of VST plugins is when the DAW takes control of the inner parameters of plugins. But these parameters can also be edited from within the plugin. In the case of conflict DAW has the priority and the editors should reflect the changes in parameter values at all times. In the case of our plugin all synthesizer parameters are automatable.

Providing automation can easily become a complex problem, with many cases to consider. JUCE provides the complete solutions for it in a simple and robust manner. We were impressed how easy it was to implement automations.

3.1.2.2 Value Trees

Value Tree's are JUCEs built in flexible data structure. I provide easy solutions for saving and loading data and undo/redo operations. It also has listeners implemented that can receive information about what has changed in the tree. It was hard to grasp how to use them at first but in the end we have used their save/load and listener functionalities extensively.

There are also some downsides to Value Trees. Properties of ValueTree's are created and accessed through string identifiers which reduces the error catching capability of the compiler. Modifying and reading ValueTrees are also not thread safe and we needed to implement locks to ensure program stability.

3.1.2.3 Threads and Locks

The plugin template in JUCE by default works with multiple threads. JUCE provides some easy to use locks to do operations in a thread-safe manner. We relied on Read Write Lock extensively.

3.1.2.4 Filter

We have used a filter implemented by JUCE in the synthesizer. It was relatively easy to implement and worked nicely.

3.1.2.5 User Interface

User interface in JUCE is built with classes that are called components. Components can form hierarchies (parent child relationships) with each other. They have default callback

functions such as resized and paint, in which we can flexibly implement the behaviours we want. Components classes can be extended with listeners to react to things like mouse events and changes in value tree data. Basic GUI elements such as buttons and combobox are really easy to use. There are also more complex classes such as viewport that are easy to use.

Overall the user interface functionalities of JUCE are simple and easy to understand use but incredibly flexible. This flexibility aided us in making our unique application. But there were also some aspects of the user interface that were hard to use.
During the implementation of the UI, we started by hard coding the component's placement to easily proceed at the start. The component places can still be made reactive within the resized() function but this resulted in a code that is hard to understand, write and modify.

We decided to refactor this UI code and after some research we found two juce classes which are called Grid and FlexBox. When you read their documentation you can see that they state that these classes are implemented using the CSS references. One of us was already experienced with CSS, so he designed the structure on paper and later started to implement it. But during the implementation we've seen that you can't put grids inside another grids; nested grids were not allowed. Also behaviour of the flexbox was not as we expect. At the end instead of that many flexible classes, a basic grid look with no-nested grids are used.

We wanted to implement a window with a progress bar that will increase while the system is placing the node and edge data after a MIDI file is read. We wanted to implement this feature on a different thread. After hours of work, the problem we were facing was obvious but not solvable. When we want to increase the percent ratio of the progress bar, we need to work on the message thread of the juce, which might be thinked as a UI thread. But we wanted to read the data on a different thread, and juce required us to lock the message thread to block any kind of threading problem. So at the start due to the heavy load of placing nodes UI was not incrementing and at the end we needed to lock this incrementing thread so again it was not showing the progress to the user. We ignored that feature.

We have tried using the Lasso Component class of JUCE for multiple node selection. But the documentation was very unhelpful and the class itself was only half implemented where many basic features were left to the user. At the end of the day it was much easier to implement such basic functionality from scratch.

3.1.3 Trello

During the implementation we've seen how strong of a tool trello is. We used 5 main columns, which are Backlog, To Do, Working On, Done and Off List. Backlog was containing all the tasks we thought as necessary at some point, and we were moving

the tasks from backlog to to do as we decide to implement them. The person who got the task was assigning himself to the task so that others could see who was working on what. Working on and Done columns are obvious. Off list contains the tasks that might be implemented at some point but not necessary in near feature. For example a task in the backlog might be tempo change, while a task in the off list is being able to specify custom colors to nodes. With this mindset we tried to get as many tasks as we could and proceed like that. If the number of tasks in the done section passes the ten-fifteen range, we archive that column and create a new column with the same name. This cleans up the space in Trello.

3.1.4 Optimizations

At a composition, there can be many nodes and edges. This is why we had to optimize the data of the composition somehow to avoid O(N) complexity. We did so by dividing the composition to time windows. In the implementation, as of writing this report, time windows have a length of 2 bars (8 beats). For every unit of two bars in the composition, a reference to every node and edge that occurs within those two bars is put into an array that we call a window. When the processor is searching for which nodes and edges to play in playback it is only checking two windows at a given time. The time window updates are thread-dangerous and thus requires locks.

The note editor would also run slow if it had to draw all nodes and edges. The optimizations of the note editor are provided mostly by JUCE where invisible parts of the editor are not drawn.

3.1.5 The Synthesizer

The synthesizer part of the program is something that we spent little time on, but surprisingly turned out great. As the source of all the sounds the program can emit, no matter how easy to use and powerful the note editor is, our program would not be able to shine without a good synthesizer. The synthesizer has a minimal set of features but together they can create very diverse sounds ranging from soft and small to noisy and majestic. Our prior knowledge about how synthesizers and sound signals work has helped us a lot in the project. Most features of the synthesizer are common features that can be found in other synthesizers. But the synthesizer also controls the duty cycle of square waveforms and slope of triangle waveforms (at maximum slope they become sawtooth waveforms). These features are usually nonexistent in analog or software synthesizers, but are a part of early computer chip synthesizers. We also included them because they were easy to implement and sounded great, even though their effects on sound somewhat overlapped with filters. The filter we have used in the synthesizer is the State Variable TPT Filter implemented by JUCE.

3.1.6 Final Weeks of the Project

Before the end of the project we wanted to bring our program to a state where it can be reliably used for fantastic sounding compositions. This is why we have used our

last few weeks bug fixing, increasing stability, implementing features that are necessary for the ease of use and making synthesizer sound good. Instead of going for new and big ambitious features.

In order to make sure that the program is stable, usable and extensible we also opted out from some features and bug fixes that are not so hard to implement but would require additional testing and/or be much better if they are done in later stages of the project. One of these features is allowing different time signatures and also being able to configure the time grid (and time snap) density. Another one is a bug that happens with ableton when the plugin editor might get loaded sooner than the saved state of the processor. This gets in the way of the editor reflecting the data in the processor correctly but can be fixed by simply closing and reopening the editor once. This will naturally stop being an issue when we implement undo and redo at some point. So we didn't fix it for now.

5. Testing Details

The main testing activity in the development was using the program ourselves. And while doing so find and fix bugs, crashes, improve the experience and optimize the code until the program becomes up to our standards.

5.1 Debug Tools

For debugging we relied on built in tools of Visual Studio. We also linked the debugger to two different DAW's (Ableton and Reaper) in order to debug it as it's running in VST hosts. The debugger allowed us to easily identify our mistakes and unexpected scenarios where our logic failed.

5.2 Edge Cases

For most of the testing part we thought about the edge cases, and applied them manually. There were not one but 3 different bugs that occurred because of floating point numbers being exactly equal to each other.

5.3 Stress Testing

We also applied stress testing to evaluate performance and optimizations. By using the copy and paste feature of the Microgliss, we tried the maximum number of nodes and edges that we can put in a time window before lagging or any similar problem starts. That number was approximately 160 for a time window (8 beats), which is more than enough. Also by opening multiple instances we've seen that multiple instances can be run on a DAW which is a desirable quality for audio plugins.

5.4 User Experience

For the user experience part, we need to mention that ⅔ of our developers are also hobbyist musicians that frequently use music software, tools and have experience with composing. So for most of the project, it was easy for us to change perspective and look from a musician/user side because we were literally using the tool to compose as we developed it.

For example for the zoom in/out functionality, we reached an adequate quality. It was zooming in and out and it was close to where you are pointing. From a technical perspective that would probably be a pass, but as we are also end users, we decided that being really close is not enough because after 5 minutes of use, it was starting to become tiring. So we spent more time on that functionality and found the underlying problem for the lack of precision, and solved it.

5.5 Beta Testing

At last we've created a discord channel and invited people around us that are capable of testing the product and creating music with it. The main purpose of that was applying a little beta testing, alpha testing was done by the developers and inspecting non-developers in a controlled environment might be hard in pandemic conditions. But we wanted to make the product as good and stable as it can be before starting a beta phase. In the end we couldn't start the beta process in time and so we couldn't get any feedback which might be useful for this report. But the program is now in a state where we would be comfortable with conducting some beta tests.

6. Maintenance Plan and Details

It might seem like we implemented almost all the functionality we've promised, and also added more than we've planned; but there are still lots of things that we want to add. We really believe in what this project is capable of and still have lots of ideas we want to implement in the later stages, such as assigning instruments to each node, so that their sound is also interpolated. We will keep using Trello for new features, and will keep in touch with other teammates.

For the maintenance plan, we already had a discord channel which people can share any problem they are facing. Problem will be talked in detail and will be added to our issue list in Trello, and will be fixed shortly.

Other than that, Microgliss is a program that works on its own without need of any other systems, such as a server. Its basic maintenance only covers bug fixes, if there is any.

Also audio software technologies progress in a much smaller phase compared to the rest of the software industry. Thus, in its current version the program should be supported for a long time.

7. Other Project Elements

Microgliss also has details and elements to it that are not related strictly to code. Most of these important factors will be discussed in the following subsections.

7.1 Consideration of Various Factors in Engineering Design

Microgliss followed an engineering procedure, including asking the opinions of innovation experts and people well-versed in the area of music. We ourselves were knowledgeable regarding this area, which made things easier, but also opinions of others affected our motivation for this project. Reports we have made previously gave us opportunities to think about and plan both what we wanted from the program and how we would go about implementing them. This planning and thinking helped us when selecting paths and approaches in the actual implementation.

Though, we were also not hesitant to refactor our code and/or change our approach whenever we felt that it was necessary. The way note and edge data is held and manipulated is a main backbone of the program with a lot of systems acting on it and depending on it. The parts regarding the storage and usage of this data is rewritten a couple times to ensure that it is providing everything we request from it, including stability and performance.

7.2 Ethics and Professional Responsibilities

Even during its inception, we have considered the ethical implications of Microgliss, a software intended to be used by musicians worldwide. Music is present in innumerable cultures, which means Microgliss will be involved with them and naturally it must not infringe upon any rights or cause harm to people’s values. Also people which will be using it will be musicians which might not have computer-wise knowledge. And due to its creation, Microgliss should support creativity and ensure that rules are there to be broken.

We can summarize our responsibilities as:

- It should be culturally appropriate.
- It should be easy to use for any musician who has used a music plugin.
- It should have room for creativity.

7.3 Judgements and Impacts to Various Contexts

We believe that ease of use in microtonal editing and glissando writing that Microgliss will provide will have a large impact on digital music making, and in the musical industry in general. As for the judgements we made, at all stages, we utilized

our knowledge regarding the music area and also naturally from an engineering standpoint that we obtained with our education at Bilkent University. Our design choices were influenced from our past experiences and the material we were exposed to at Bilkent University.

7.4 Teamwork Details

In any modern project teamwork is crucial for the project's success and quality. Without cooperation, Microgliss would not have been possible. The following subsections will list the most relevant aspects of teamwork that shaped this project.

7.4.1 Contributing and functioning effectively on the team

Thankfully, no major disagreements were present during the project, and the members mostly fulfilled the tasks they were given with high effectiveness.

7.4.2 Helping creating a collaborative and inclusive environment

All members were helpful to one another during the project and encouraged teamwork and collaboration. This sped up development and made for high quality of communication between team members.

7.4.3 Taking lead role and sharing leadership on the team

When it came down to it, every team member knew how to take leadership in a certain area or phase of the project, allowing development and progress without a hitch.

7.4.4 Meeting objectives

As mentioned before in the report, Microgliss changed into a different form than what we initially anticipated. This caused some objectives to become either obsolete or change focus. Regardless, these new objectives, if any, were mostly met along with our original goals during the development of Microgliss. There are still some more functions we would like to implement on Microgliss, but these are mostly either quality of life changes or ideas we came up with during development.

7.5 New Knowledge Acquired and Applied

We have learned:

- How to use the JUCE framework
- Different memory management strategies in C++
- Intricacies of creating an editor application
- How VST plugins are worked
- How VST plugins can be developed

- How to build our own Synthesizer
- How to use Trello effectively in a software project
- How assumptions doesn’t hold sometimes

We have practiced making a program thread safe using locks. We have also come to appreciate how much work is involved in making an editor application stable, optimized and up to standarts.

Also from the soft skills side, we’ve seen how another teammate might motivate you to keep working even though you are facing a problem.

8. Conclusion and Future Work

We have consciously chosen the scope of our project to be small. But this doesn’t mean implementing was easy. Even though the scope is small, it goes into low levels of audio processing and manages complex data in order to achieve the things it does. We have put so much effort in it to make sure that it is up to standards and usable by musicians in order to make great compositions that are like nothing else.

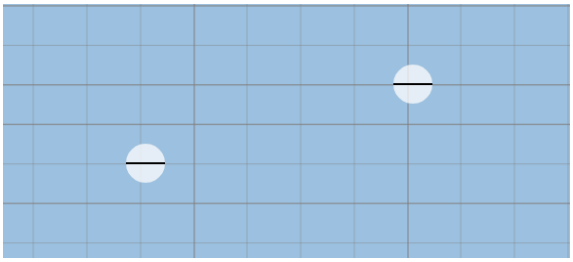
Our focus has started on microtonal music, but later appreciated that polyphonic glissandos and flexible note velocities that are enabled by our workflow are as important, if not more important than the microtonal capabilities of the program. In fact, our workflow has become much more powerful, fun and intuitive than we could have imagined. We wish that many musicians will use Microgliss and like the new way it introduces to making music as much as we do.

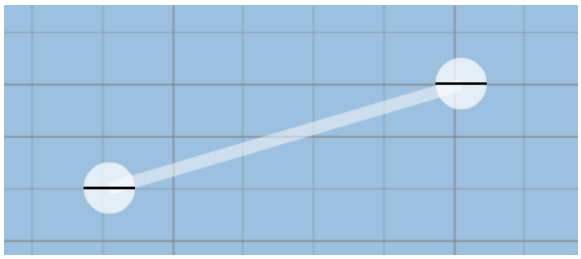
microgliss
user manual

Contents

Introduction

Microgliss is a digital synthesizer with unique note editing capabilities. It includes an **editor screen** which you can control what will be played, and a **synthesizer screen** that you control how it will sound like. In the editor, there is an area with a grid which you can add circles which represents notes and connect these circles using lines.





Lines get played continuously, so by having two circles at different positions and connecting them with a line causing the frequency of the note to change seamlessly. This type of note editing is the core difference of microgliss. After this point these circles will be referred to as **nodes**, and lines will be referred to as **edges**. This terminology comes from graph theory in computer science. Also during the manual edges and nodes will be referred to as **nodges** if any functionality is common.

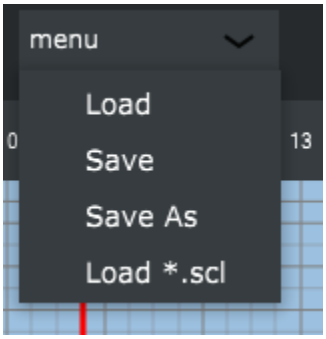
This manual focuses on microgliss. You might need to check other sources to understand not microgliss specific information such as what frequency, amplitude, ADSR etc. is.

Common Items

Top Bar

Top bar contains a **dropdown menu** and an **interface selector** which consist of two buttons for switching between the editor and synthesizer interface of microgliss.

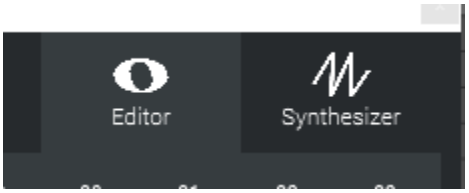
Dropdown menu



Load : Loads composition in *.midi or *.mglc form. Mglc is the form microgliss use for saving data. `Ctrl + L`
Save : Save your composition as *.mglc type. You will select a file name when you save a file for the first time. After that, saving with this button. `Ctrl + S`
Save as : Save your composition as *.mglc type. You will select a file name every time you save a file.
Load *.scl : Loads a scale file which changes the grid lines according to the tuning format specified in the scale file.

Interface Selector

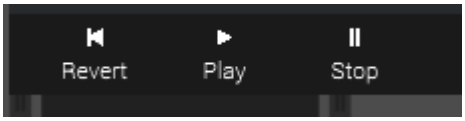
With the interface selector buttons you can switch between note editor and synthesizer screens. Enabled button will light up and indicate that button is clicked.



Bottom Bar

Bottom bar consists of a group of playback control buttons, copy/paste buttons, beat and cent locks to snap the nodes to the horizontal or vertical axis and a master knob which can control the overall volume.

Playback Control Buttons



Revert: Sets the playback bar position to zero (The leftmost position on the grid).
Play: Starts the playback. If microgliss lives in a host, bpm settings will be the host's bpm.
Stop: Stops the playback bar and any sound that microgliss was playing at the time.

Copy/Paste buttons

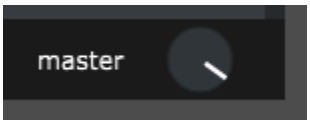
Copy: Copies the currently selected nodge(s).
Paste: Pastes the currently copied nodge(s).
Copy and Paste can be done together at once by `holding Ctrl while dragging a node`.

Beat & Cent Locks

- Beat Lock: A toggle button. While toggled on, the picture on it will become a locked lock, and the program will attempt to create or drag nodes exactly to the closest beat start grid line.
- Cent Lock: A toggle button. While toggled on, the picture on it will become a locked lock, and the program will attempt to create or drag nodes exactly to the closest pitch grid line.

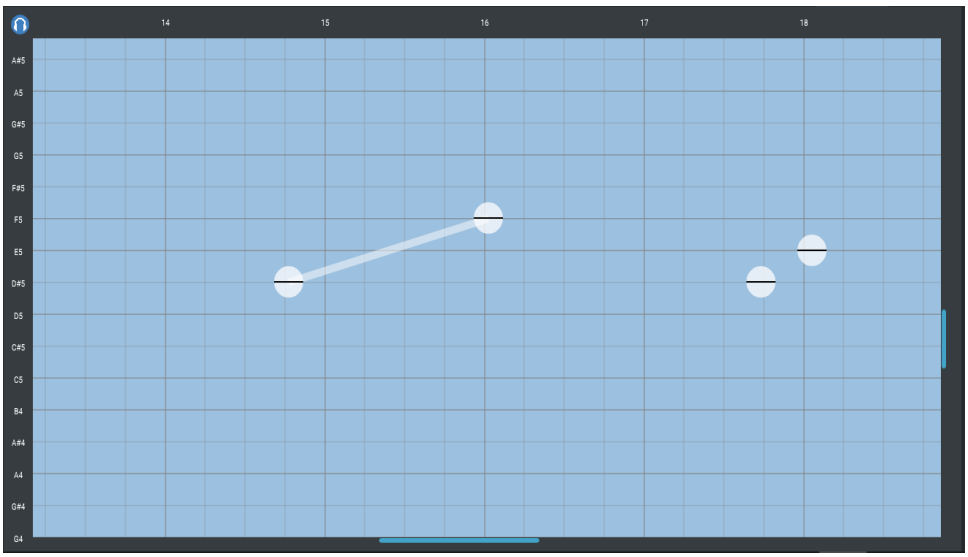
If any of the locks are enabled and nodges are being dragged, nodes will be snapped to the related closest line.

Master gain



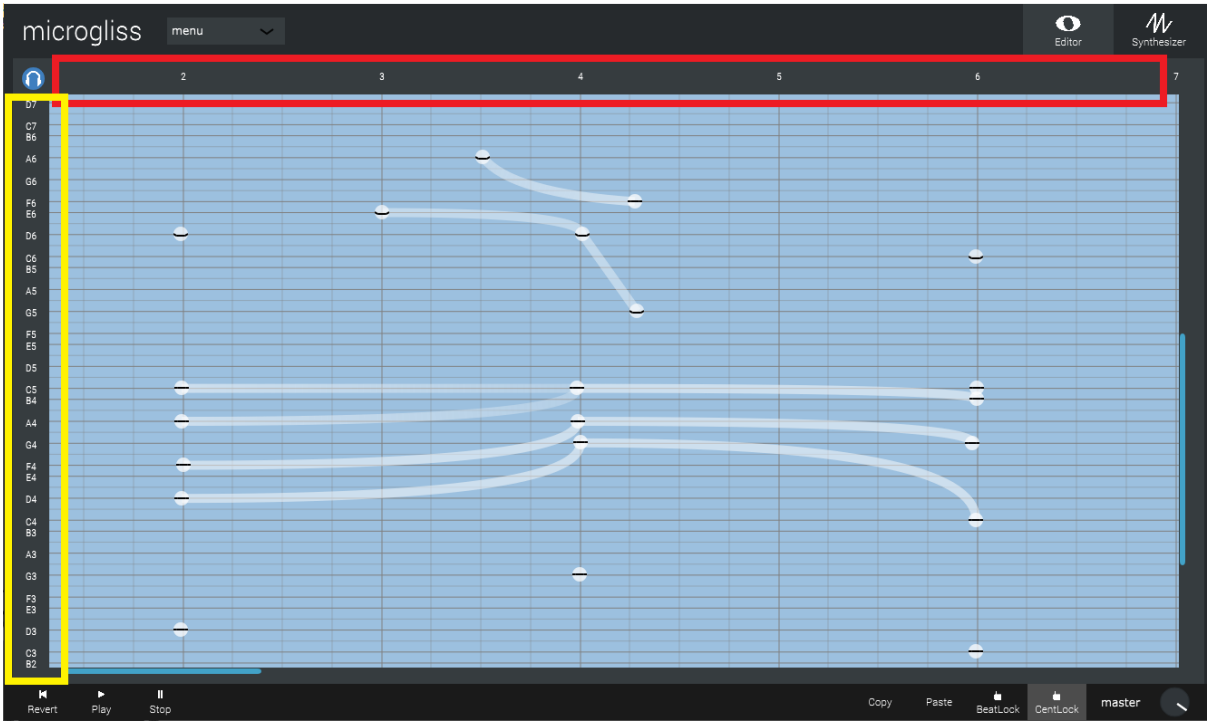
Controls the overall audio output of the plugin.

The Note Editor



The note editor consists of an outer frame and a grid area. In the whole note editor, the vertical axis represents the pitch of the notes, and the horizontal axis represents the time.

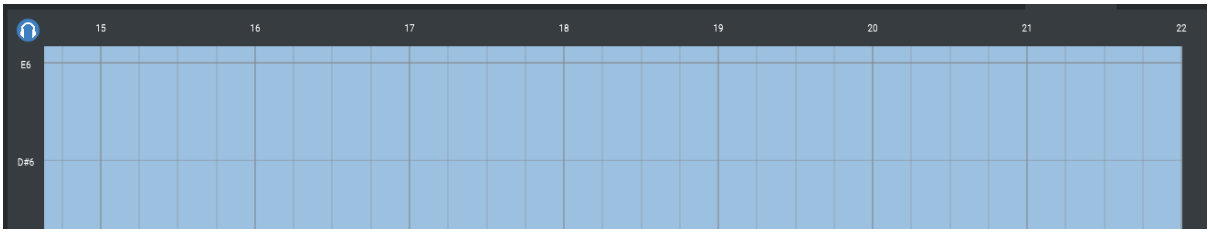
Outer frame

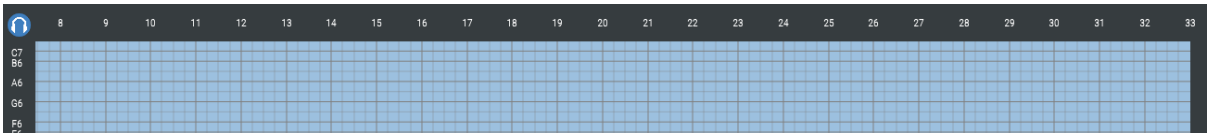


On the left hand side of the frame (yellow) you can see the details of the pitch information,and the upper part (red) contains information about time data, in beats. And in the middle there exists the grid. To make changes in the grid:

- Scroll to move up/down vertically
- Alt + scroll to move left/right horizontally
- Ctrl + mousewheel to zoom in/out vertically
- Alt + mousewheel to zoom in/out horizontally

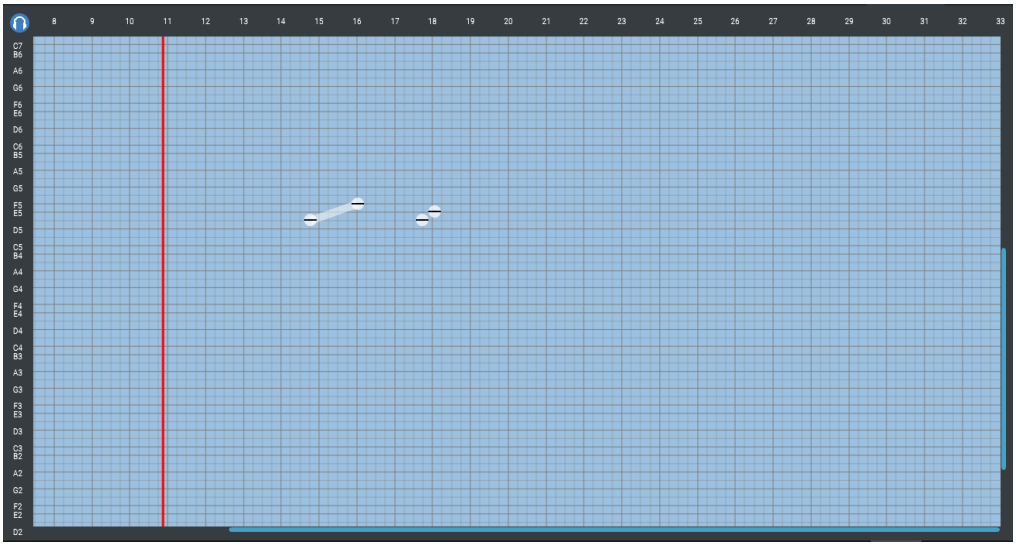
Outer frame gets updated according to the changes in the grid.



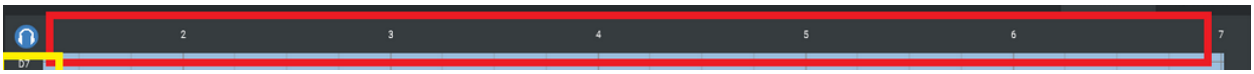


Also, the headphone button on the top left portion of the outer frame indicates that aural feedback is on, and notes will be played back instantly on input. This might be distracting sometimes, so you can `disable the feedback by clicking on the headphone button`.

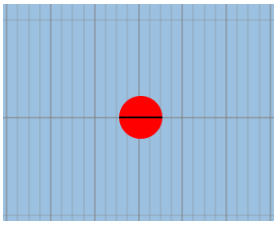
The Grid



The grid has horizontal lines indicating the pitch information and vertical lines that indicates time information. Pitch information lines of the grid can be changed by `selecting a different *.scl for grid from the menu` in the upper bar as indicated in the **Dropdown Menu** section.

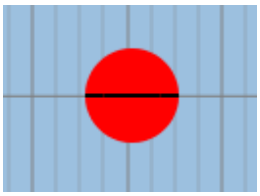


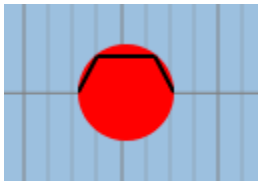
The grid has a vertical red bar indicating playback position on it. You can change the position of the playback bar by `left-clicking on any time value on the horizontal axis on the outer frame`. DAW playback always has priority over the plugin's inner playback.



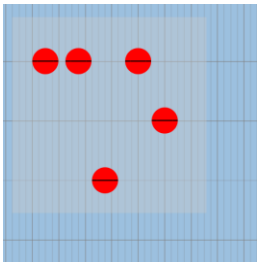
You can place nodes on the grid by left-clicking on an empty area, and a node will be created with middle level velocity at the location the you clicked.

Node velocity can be edited by dragging the horizontal black line along the node up or down to raise or lower it, respectively. It is possible to edit the velocities of multiple nodes at once.

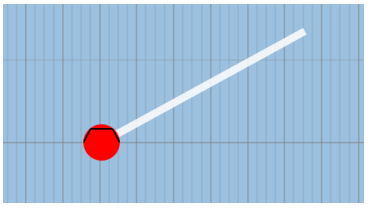




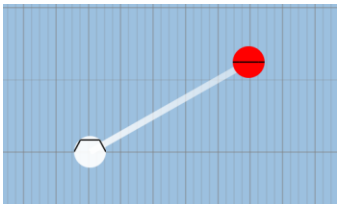
You can select multiple nodes by right-clicking to a point and drag their mouse while holding down. This will create a transparent rectangle that indicates the selection area.



Holding Ctrl keeps selected nodes selected, even though the mouse is released. This lets you select multiple nodes without letting any node go.



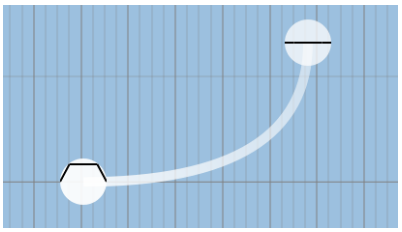
Once you left-click an existing node or create a new node, that node will be selected and colored red. Then, you can hold Left-Shift to attempt to create an edge. While holding Left-Shift, if you left-click anywhere on the grid, If there is a node in the newly clicked position form an edge between the newly clicked node and the previously selected node. If there isn't a node in the clicked position, Microgliss will create a new node at the clicked position and form an edge between the previously selected node and the newly created node. It is also possible to select multiple nodes and create multiple edges starting from them and ending in a position at once.



You may also select all nodges by pressing Ctrl + A.

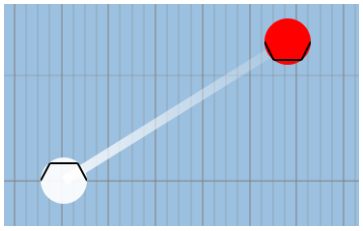
You can delete all select all nodges by pressing Delete.

You can `delete a single nodge with a double click.`

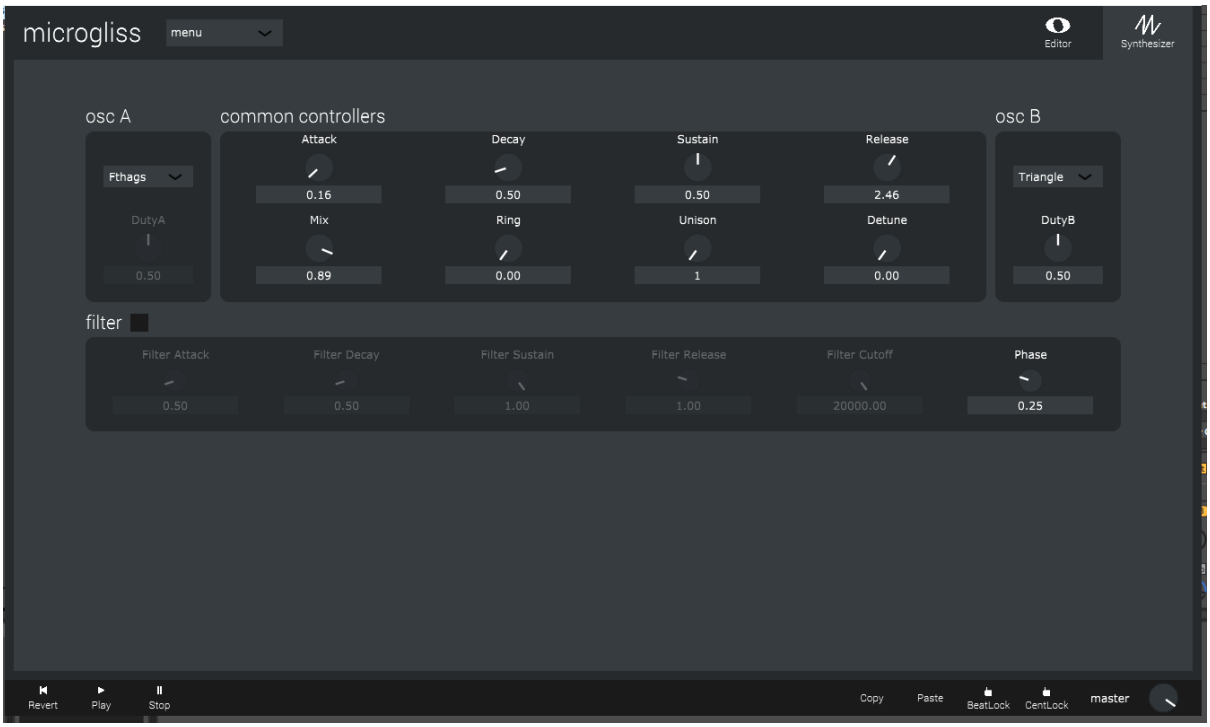


You can click on an edge to select it, then drag up or down to give a curvature to the edge. The way that frequency change follows the curvature of the edge. It is possible to edit multiple edges at once.

Edge velocities are represented by the gradient of their coloring, with fading colors indicating lower velocity, and brighter colors indicating higher velocity. Edge velocity depends on the velocities of the two nodes it is connected to.



Synthesizer



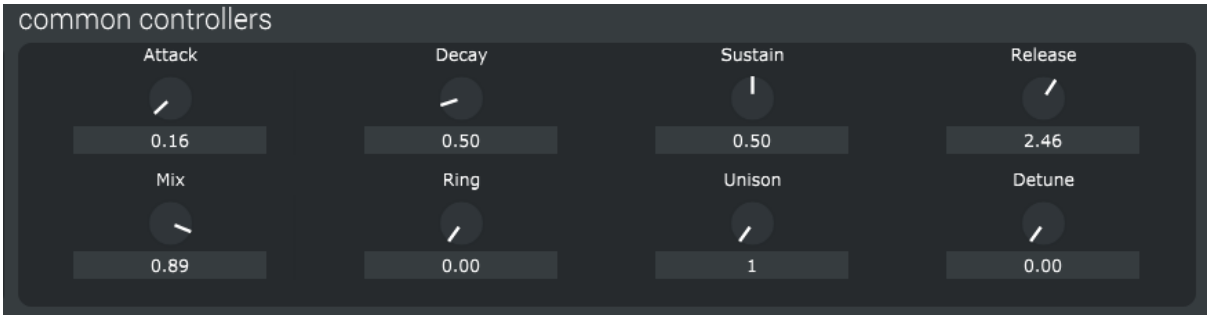
This is the synthesizer for microgliss. It is minimal in design and features but you can create many sounds with it. Ranging from small and cute to majestic and noisy.

osc A & B

Oscillator A & Oscillator B both have A waveform selection box and a duty knob. Waveform selection box provides 5 basic wave options:

Sine, Square, Triangle, Noise and Fthags. Duty only affects triangle and square waveforms, and changes their timbre. (Technical details: Depending on the slider value, duty of the square wave ranges from 0 to 0.5. And at 1.00 triangle becomes a sawtooth wave).

Common Synth Controllers:



Attack, Decay, Sustain and Release: Use it to edit the ADSR envelope of the volume. This determines how loud a note will be at the start, middle and end of its lifetime.

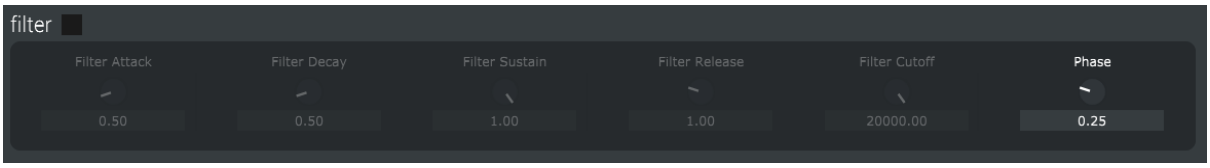
Mix: Mixes osc A & osc B (At 0.00 it's %100 oscA, at 1.00 it's %100 oscB).

Ring: Applies ring modulation (multiplies the signals instead of adding them).

Unison: Changes number of voices for a nodge. Up to 16 voices are allowed. (Using too many voices might reduce the performance).

Detune: Detunes the unison voices, creating a rich phasing sound.

Filter



Filter can be enabled/disabled with the button provided right next to the filter.

Filter Attack, Filter Decay, Filter Sustain, Filter Release: Use it to edit the ADSR envelope of the fılter, determining how much the filter will be applied over the lifetime of the note.

Filter Cutoff: This value controls the maximum cutoff frequency of the filter. Lowering this will cause the sound to get softer, darker, and/or more muffled.

Phase: Controls the phase of the two waveforms. Leaving it at 0.25 should be fine for the most of the time and it makes minimal effects to the sound. Be careful, if the value is 0.50 (or 0.00 with ring modulation) waveforms can eat each other resulting in silence.

*Phase doesn't have anything to do with filters. (We couldn't find any place to put it for now :P)