Bilkent University

Department of Computer Engineering

Senior Design Project

microgliss: a microtonal editing tool for the world music

Low-Level Design Report

Group Members: Artun Cura, Sonat Uzun, Orkan Öztrak

Supervisor: Vis. Prof. Dr. Fazlı Can

Innovation Expert: Burcu Coşkun Şengül

Analysis Report
February 8, 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.

Introduction

Microgliss is a note editor and synthesizer that allows you to write microtonal music with polyphonic glissandos using an intuitive node and edge-based workflow. It will allow users to write music in any tuning system or independent of any tuning system. It will allow users to add glissandos between any notes. Microgliss is designed keeping in mind the growing interest in microtonal music around the globe and the common limitations of existing midi/note editors and synthesizers. Using midi editors is a common method for composing computer music. But MIDI editors have limitations that can't be solved or only can be solved with non-elegant workarounds. Some of these limitations are 128 different values for velocity, 128 different note values (pitch), once a note's value and velocity are set they can not be changed until that note is released, enforcement of a grid system for pitch values (as a result of 128 note limitation), no glissandos for multiple notes, etc.

World music is not constructed on a grid. Every culture has its own unique sounds, pitches, and scales which emerged from freedom. But today's computer systems make the production of world music(non-western music) harder for the users because all of the systems are constructed around western culture. We are trying to solve these problems with microgliss.

Object design trade-offs

While developing and designing software, developers need to sacrifice a part of the functionality to enhance some part of the other functionality. This process is called object design trade-off and in the following sections, the trade-offs we have encountered during the development of microgliss will be presented and explained.

Functionality vs. Usability

Microgliss' aim was to offer solutions to some problems which current systems bring-along. We tried to keep these new solutions as simple as possible user-wise to ensure that they don't encounter too many new things and get confused.

When you inspect a musical entity such as a basic sine wave, it is easy to see that it has many parameters such as amplitude, frequency, phase, etc. and these parameters are easily manipulatable; but also it shows that when we have multiple entities, letting every parameter get controlled by the user might result in the loss compactness and clarity of the program pretty quickly. To eliminate that, we tried to choose usability over functionality.

Compatibility vs Extensibility

At the start of the development process, we chose the JUCE Framework which lets us export our program in VST(Virtual Studio Technologies) and AU format, a format most of the current digital audio workstations support. The reasoning behind this decision was making our program compatible with almost all of the Digital Audio Workstations and not caring about the details about the compatibility team-wise.

We wanted to invest our time on developing new features and enhancing the existing ones, instead of trying to export in more formats. So we can say that extensibility is more important for us than compatibility because that function is handled by the framework we chose.

Cost vs. Performance

Digital Audio Workstation users usually work with many channels(sometimes up to 200 channels), which means there might be multiple microgliss instances that produce sound simultaneously in addition to other plugins and recorded instruments. If these channels use the CPU heavily and require more processing power than they should, real-time listening functions of the DAW's start to stutter and give a glitchy sound. This is why a music plugin should work as efficiently as possible. This is also the reason why we choose performance over the cost in the development of the microgliss.

Interface documentation guidelines

In this report, we named classes as "ClassName", variables as "variableName" and methods as "methodName()". They might be considered as the standard way of naming these components. The overall hierarchy starts with the class name, followed by its purpose and explanation, then its attributes, and methods.

class ExampleClass	
This class is an example for this report	
Attributes	
private string name private int age private bool hasGraduated	
Methods	
public string getName() public void setAge(int age) public void setGraduated(bool graduated)	returns the name of the object sets the age attribute. sets the hasGraduated attribute.

Engineering Standards

During the reporting of our senior project, we used UML Guidelines [1] to define class interfaces, diagrams, scenarios, use cases, and subsystem compositions and hardware. For the citations, the report follows IEEE's [2] standards. We chose UML and IEEE because both of them are commonly used ways to generate diagrams, and every member of our team knew it.

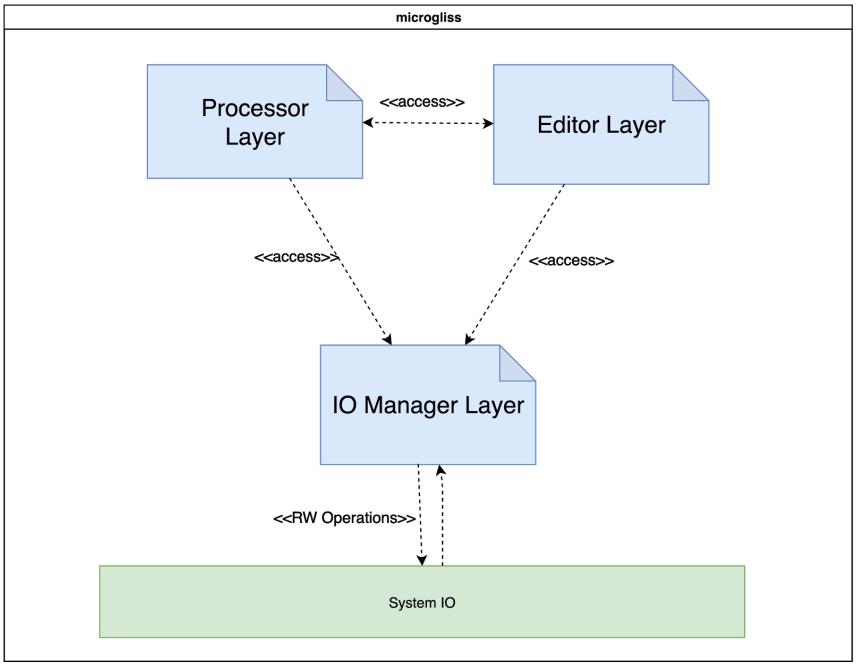
Definitions, acronyms, and abbreviations

All definitions, acronyms, and abbreviations used in this report are explained in the glossary section.

Packages

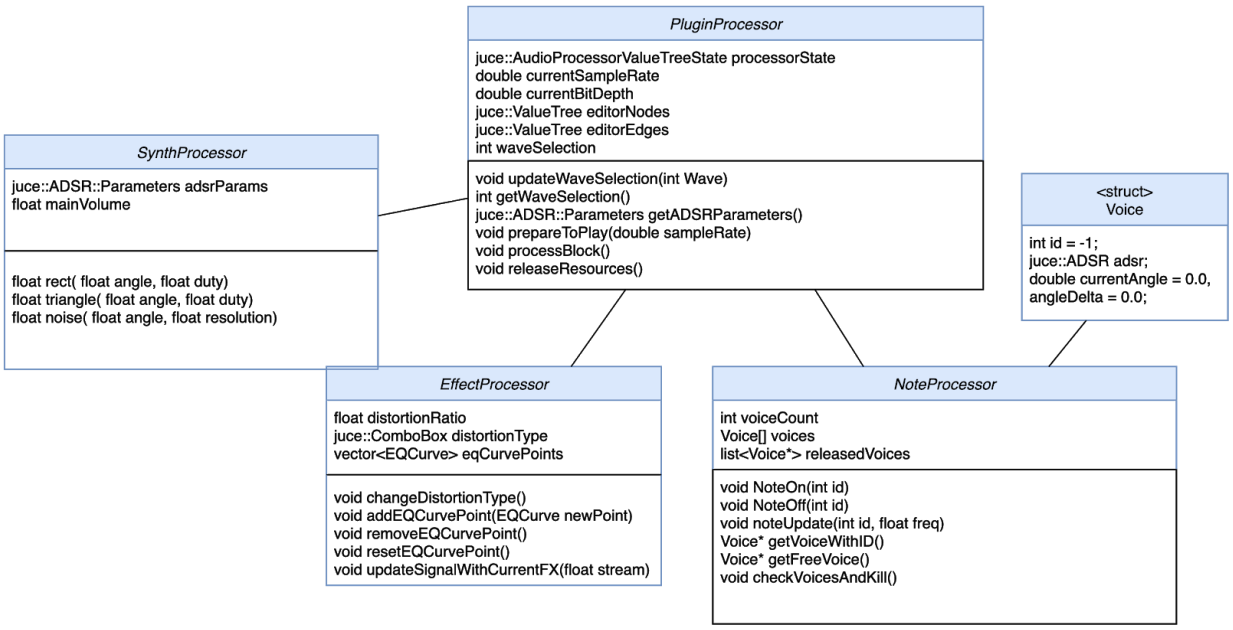
Microgliss is composed of three main layers which are Processor layer, Editor layer, and IOManager layer. This layer system is updated after the High-Level Design document.

The relation between them can be seen in the graph below.



Processor Layer is responsible for sound generation, signal transmission, and in-app data management. This layer has *PluginProcessor*, *SynthProcessor*, *EffectProcessor,* and *NoteProcessor.* The second layer is the Editor Layer. It controls most of the visuals, buttons, and some basic operation logic. This layer has *PluginEditor, NoteEditorFrame, NoteEditor, EditorNode,EditorEdge* and *SynthesizerEditor*. The third and last layer of microgliss is the IOManager Layer.
This layer is responsible for the filesystem operations such as save/load or import/export. This layer has *SaveLoadManager, DAWCommunicationManager, MicrotonalFileImportManager,* and two little structs *sclDataPack* and *tunDataPack* for external tuning system import.

Processor Layer



PluginProcessor

A class that is responsible for sound generation and buffer operations.

SynthProcessor

A class that contains synthesizer parameter data and synth-related sound generation functions.

EffectProcessor

A class that contains the basic effect operations such as distortion and EQ. Also holds their data.

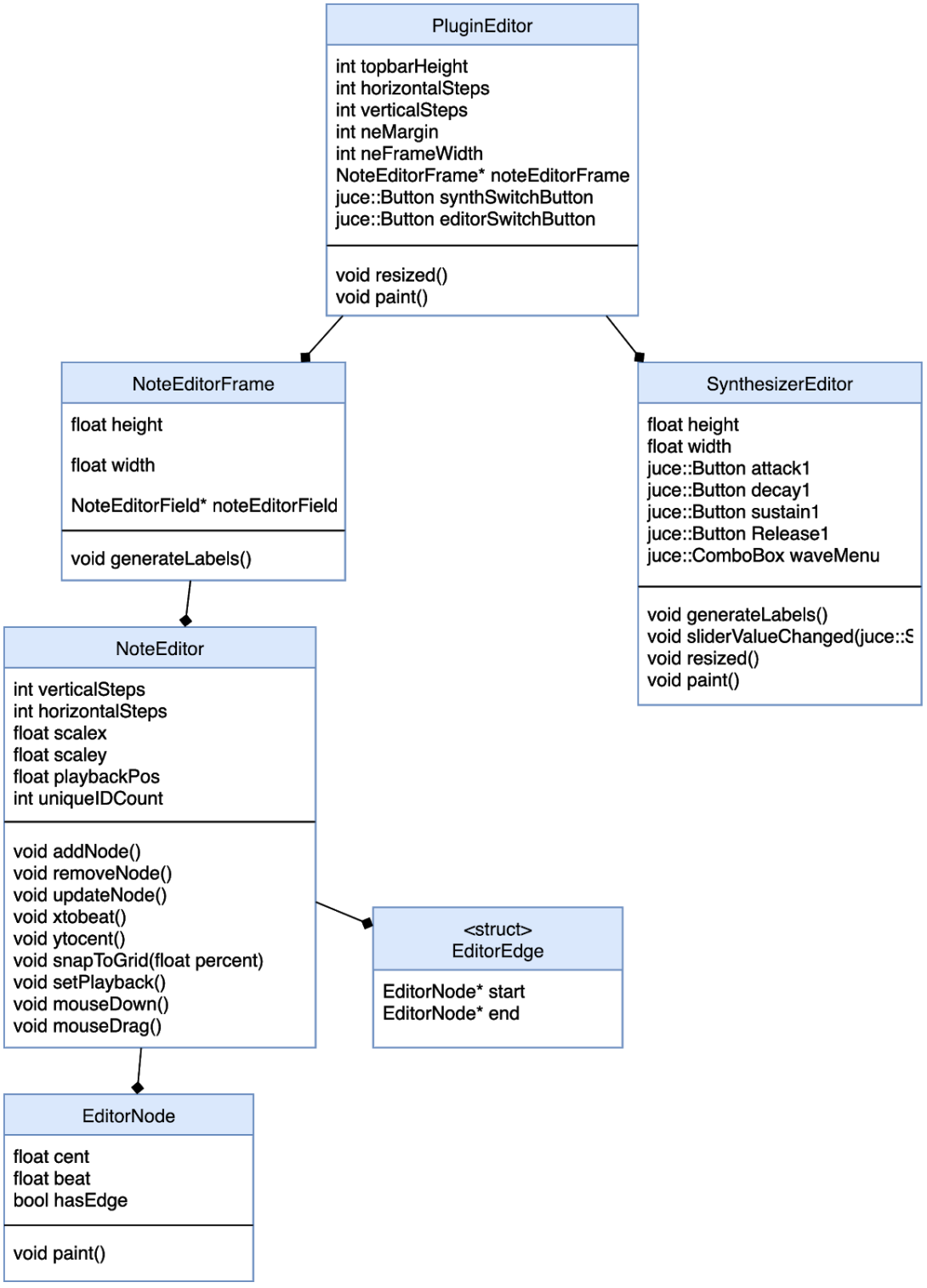
NoteProcessor

A class that contains the voices and note related operations.

Voice

A struct with ADSR and frequency information.

Editor Layer



PluginEditor

A class that contains visual components of the outer plugin frame. That frame contains the logo, tuning file import operation button, and editor/synth selection buttons.

NoteEditorFrame

A class that contains components of the outer editor frame. These components are beat and cent information.

NoteEditor

A class that contains the nodes, edges and their add/remove/modify operations.

SynthesizerEditor

A class that contains the ADSR options and wave settings of the microgliss' synthesizer.

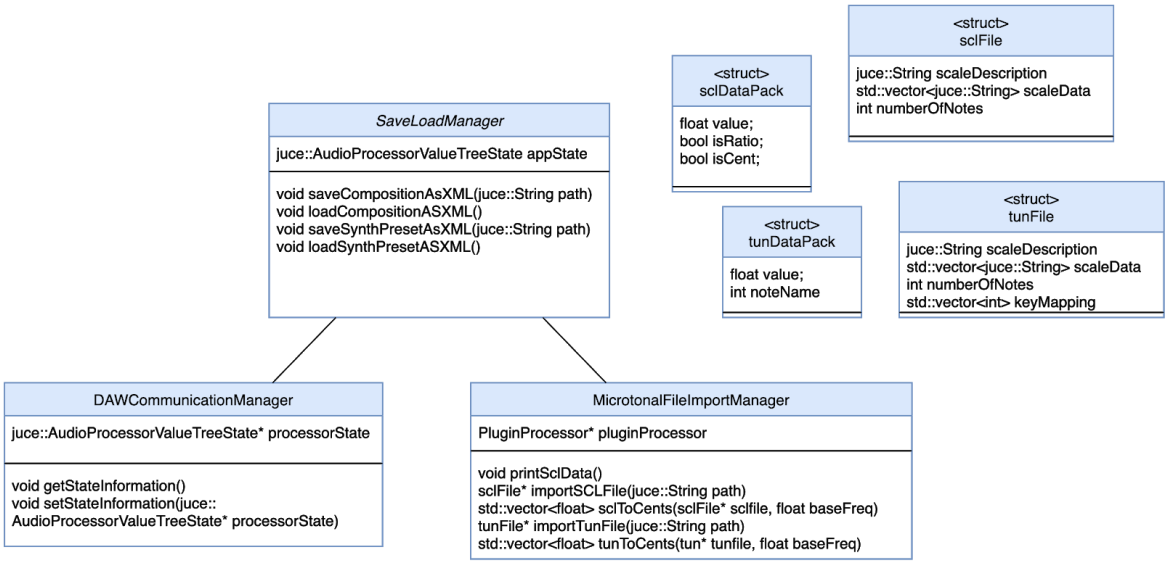
EditorNode

A class that contains the node data and controls the visuals.

EditorEdge

A class that contains the edge data.

IOManagerLayer



DAWCommunicationManager

A class that manages the transmission of the information such as playback location and sound output between the plugin itself and DAW.

SaveLoadManager

A class that manages the save/load operations of microgliss' editor and synthesizer.

MicrotonalFileImportManager

A class that controls the microtonal file import operations for the grid layout.

sclDataPack

A struct that holds the fundamental scl tuning data.

tunDataPack

A struct that holds the fundamental tun tuning data.

sclFile

A struct that holds the scl description, number of notes and tuning data.

tunFile

A struct that holds the tun description, number of notes and tuning data.

Class Interfaces

Processor Layer

Plugin Processor

class PluginProcessor
This class is responsible for sound generation and buffer operations.
Attributes
private juce::AudioProcessorValueTreeState processorState private double currentSampleRate private double currentBitDepth private juce::ValueTree editorNodes private juce::ValueTree editorEdges private int waveSelection
Methods

public void updateWaveSelection(int Wave) public int getWaveSelection() public juce::ADSR::Parameters getADSRParameters() public void prepareToPlay(double sampleRate) public void processBlock() public void releaseResources()	changes the selected wave returns the selected wave returns the current ADSR parameters struct prepares the plugin to play sound processes the audio block releases dedicated resources for audio

Synth Processor

class SynthProcessor
This class contains synthesizer's parameter data and synth-related sound generation functions.
Attributes
private juce::ADSR::Parameters adsrParams private float mainVolume
Methods

public float rect(float angle, float duty) public float triangle(float angle, float duty) public float noise(float angle, float resolution)	generates a square wave signal generates a triangle wave signal generates a noise signal

Effect Processor

class EffectProcessor
A class contains the basic effect operations such as distortion and EQ.
Attributes
private float distortionRatio private juce::ComboBox distortionType private vector<EQCurve> eqCurvePoints
Methods

public void changeDistortionType() public void addEQCurvePoint() public void removeEQCurvePoint(EQCurve* eq) public void resetEQCurvePoint() public void updateSignalWithCurrentFX(float stream)	changes the distortion type adds an eq curve point removes the eq curve point specified in the argument updates the output signal with current FX

NoteProcessor

class NoteProcessor
This class contains the voices and note related operations.
Attributes
private int voiceCount private Voice[] voices private list<Voice*> releasedVoices
Methods

public void NoteOn(int id) public void NoteOff(int id) public void noteUpdate(int id, float freq) public Voice* getVoiceWithID(int id) public Voice* getFreeVoice() public void checkVoicesAndKill()	activates a voice releases a voice updates a voices frequency returns voice with the given id returns an unused voice terminates voices that are no longer being used

Voice

struct Voice
This struct holds voice envelope and frequency information.
Attributes
public int id = -1 public juce::ADSR adsr public double currentAngle = 0.0 public angleDelta = 0.0

Editor Layer

Plugin Editor

class PluginEditor
This class contains visual components of the outer plugin frame. That frame contains the logo, tuning file import operation button and editor/synth selection buttons.
Attributes
private int topbarHeight private int horizontalSteps private int verticalSteps private int neMargin private int neFrameWidth private NoteEditorFrame* noteEditorFrame private juce::Button synthSwitchButton private juce::Button editorSwitchButton
Methods

public void resized() public void paint()	updates components when resized draws the graphical component

Synthesizer Editor

class SynthesizerEditor
This class contains the ADSR options and wave settings of the microgliss' synthesizer.
Attributes
private float height private float width private juce::Button attack1 private juce::Button decay1 private juce::Button sustain1 private juce::Button Release1 private juce::ComboBox waveMenu
Methods

public void generateLabels() public void sliderValueChanged(juce::Slider* slider) public void resized() public void paint()	generates and assigns the note labels gets called whenever the slider value changes updates components when resized draws the graphical component

Note Editor Frame

class NoteEditorFrame
This class contains components of the outer editor frame. These components are beat and cent information.
Attributes
private private float height private float width private NoteEditorField* noteEditorField
Methods

public void generateLabels()	generates and assigns note and time labels to grid

Note Editor

class NoteEditor
This class contains the nodes, edges and their add/remove/modify operations.
Attributes
private int verticalSteps private int horizontalSteps private float scalex private float scaley private float playbackPos private int uniqueIDCount
Methods

public void addNode() public void removeNode() public void updateNode() public void xtobeat() public void ytocent() public void snapToGrid(float percent) public void setPlayback() public void mouseDown() public void mouseDrag()	adds node to the editor removes node from the editor updates the node information of a node converts x position info time data converts y position info frequency data snaps the node to the grid sets the playback position of the grid gets called when the mouse get clicked gets called when the mouse get dragged

Editor Node

class EditorNode
This struct contains the node data and controls the visuals.
Attributes
private float cent private float beat private bool hasEdge
Methods

public void paint()	paints the graphics of the component

Editor Edge

struct EditorEdge
This struct contains the edge data.
Attributes
public EditorNode* start public EditorNode* end

I/O Manager Layer

Save Load Manager

class SaveLoadManager	
This class manages the save/load operations of microgliss' editor and synthesizer.	
Attributes	
private juce::AudioProcessorValueTreeState appState	
Methods	
public void saveCompositionAsXML(juce::String path) public void loadCompositionASXML() public void saveSynthPresetAsXML(juce::String path) public void loadSynthPresetASXML())	saves the editor data as xml loads the selected editor data as xml saves the synth data as xml loads the selected synth data as xml

DAW Communication Manager

class DAWCommunicationManager	
Thıs class manages the transmission of the information such as playback location and sound output between the plugin itself and DAW .	
Attributes	
private juce::AudioProcessorValueTreeState* processorState	
Methods	
public void getStateInformation() public void setStateInformation (juce::AudioProcessorValueTreeState* processorState)	returns the processor state sets the processor state as the passed parameter

Microtonal File Import Manager

class MicrotonalFileImportManager	
This class controls the microtonal file import operations for the grid layout.	
Attributes	
private PluginProcessor* pluginProcessor	
Methods	
public void printSclData()	prints the scl data to debugger
public sclFile* importSCLFile(juce::String path)	imports and parses an scl file
public std::vector<float> sclToCents(sclFile* sclfile, float baseFreq)	converts scl data to cents data according to the base frequency
public sclFile* importTunFile(juce::String path)	imports a *.tun file and parses it
public std::vector<float> tunToCents(tun* tunfile, float baseFreq)	converts tun file to cents data according to the base frequency

scl Data Pack

struct SCLDataPack
Thıs struct holds the fundamental scl tuning data.
Attributes
public float value; public bool isRatio; public bool isCent;

tun Data Pack

struct TUNDataPack
A struct which holds the fundamental tun tuning data.
Attributes
public float value; public int noteName

Glossary

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 electronic musical instrument that generates audio signals. 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.

JUCE: An open-source C++ Framework that is used for the development of desktop and mobile applications for GUI and Audio tools.

Trello: A basic project management tool. Its main use is for Agile project development but we will also use Trello with the Iterative approach.

Note: A symbol denoting a musical sound.

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 in 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.

Beat: Rhythmic unit in music.

Bpm(Beats per Minute): Defines how fast the music will be. Higher bpm means that the music will be faster.

Cent: 1/100th of a semitone in 12-tone equal temperament tuning, which is the default and most common way of tuning western musical instruments. Cent is often used in defining microtonal scales as well as deviations of a note from tuning.

References

[1] “Unified modelling language.” https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/. [Accessed: 3- Feb- 2021]

[2] “Ieee reference guide.” https://ieeeauthorcenter.ieee.org/wp-content/uploads/IEEE-Reference-Guide.pdf [Accessed: 3- Feb- 2021]