# O.J. DISTORTER REPORT

Table of Contents

[O.J. DISTORTER REPORT 1](#_Toc93511657)

[1. Introduction 1](#_Toc93511658)

[2. User Manual 2](#_Toc93511659)

[3. System Documentation 3](#_Toc93511660)

[3.1 The GUI 3](#_Toc93511661)

[3.1.1 Meter and Bubble Animations 4](#_Toc93511662)

[3.1.2 The Oscilloscope 4](#_Toc93511663)

[3.2 The DSP 4](#_Toc93511664)

[4. Conclusions 5](#_Toc93511665)

[4.1 Achievements 5](#_Toc93511666)

[4.2 Setbacks and Solutions 5](#_Toc93511667)

[5. Future Developments 6](#_Toc93511668)

[6. Appendix 6](#_Toc93511669)

## Introduction

The O.J. Distorter is a distortion application that takes inspiration from many popular VSTs, the rationale behind it is to present an extremely simplistic GUI while handling complex signal processing in the background, leaving the user satisfied without the necessity of a great knowledge of audio processing.

## User Manual

1

**Graphical user interface, website

Description automatically generated**

5

3

4

2

6

1. General Audio/Midi settings.

2. Play Button.

3. Rotary Slider, sets the amount of distortion applied to the signal.

4. Browse for audio files (WAV and AIFF supported).

5. Oscilloscope of output signal.

6. Background (works as Left and Right loudness meter).

## System Documentation

The overall system is divided into two macro-structures: the GUI component(s) - which are all parented by the AudioEditor class – and the audio processing component(s) – which instead are parented by the AudioProcessor class. Generally, every audio processing component which is supported by user manipulation is associated to a GUI object, all the relationships between the various classes are showed in figure 1.1.



Figure 1.1

### The GUI

The GUI presents a very simple design, and for what concerns audio processing manipulation, there is only one type of interaction: a rotary slider (of course this excludes the ‘load and play audio files’ functionality which is extraneous to the actual DSP of the application). This does not mean the implementation behind the interface is simplistic, but is simply focused on different aspects, such as design, animation and graphical representation of the signal.

That said, the key features of this part of the project are the following: loudness meter and bubble animation, and the oscilloscope.

### Meter and Bubble Animations

The meters logic is relatively simple, a function grabs the RMS value (in Decibels) of a channel and stores it into a variable which is then passed to the *MyMeter* class, where its min/max range is scaled into the min/max height of a rectangle. The rectangle is then painted (at frame rate) with a height relative to the scaled Db range, hence shrinking and growing proportionally to the overall loudness of the channel it is associated to.

The bubbles are also painted at frame rate and follow a circular pattern that lasts a second, basically they spawn at the bottom of the meters every second although they are rendered each time at a different speed and with a different radius. Again the logic is not too complex, the X, Y coordinates of the bubbles are constituted of a *juce::Point* (an object containing two float variables). The Y coordinate is augmented and updated at frame rate, while the X coordinate is randomly generated at the start of every loop, that means that the bubbles follow a straight vertical line until they disappear at the top of the application window and then spawn at the bottom at a random position. The bubbles only spawn if the slider is being dragged, that happens thanks to two *juce::Slider::Listener* member functions called *sliderDragStarted* and *sliderDragEnded*.

### The Oscilloscope

The oscilloscope implementation is more complicated than the rest of the GUI components, it happens through the work of three different classes, which work between the two application macro structures. Everything starts within the *AudioProcessor* and the *AudioScopeCollector* classes, the latter pushes data from the output buffer of the processor into the *AudioBufferQueue* class (a container of an abstract FIFO data structure), this data is then read by the AudioScope class and used to plot (paint) a sequence of lines within a bounding rectangle. The X coordinate of the starting/ending points of the drew lines are calculated by mapping the position of each sample in the buffer while the Y coordinates are calculated from the gain value of each sample.

### The DSP

The overall DSP is composed of a series of chained effects, which is displayed in figure 1.2.



Figure 1.2

Most of the DSP objects exist into the *Distortion* class (and are all composed of *juce::dsp* module classes), apart from the *Gain* object that lives in its own class and is contemporarily child of the AudioProcessor and Distortion classes. The wave-shaping function is a simple hyperbolic tangent, over and down sampling have been added to the chain in order to avoid sound artifacts and possible aliasing (as oversampling is considered a good practice whenever putting the signal through non-linear functions).

## Conclusions

### Achievements

* Stronger knowledge of the OOP paradigm.
* Much deeper understanding of the JUCE framework.
* C++ skills (a better knowledge of pointers).
* Animation Basics.
* Understanding the importance of proper versioning through GitHub.
* Understanding the importance of proper documentation (Doxygen).
* Understanding parsing of vectorial images and how to translate/import them through different softwares (Procreate, Illustrator).
* Thanks to the extensive need of going through documentation and open-source applications, a stronger capability to study and overcome problems independently.

### Setbacks and Solutions

Given that no achievements come without a good dose of effort and sacrifice, all the points above (the *achievements*) can be considered the major setbacks I have encountered during the development of this project, and I humbly take pride in saying that my stubbornness has helped me overcome them. Between all the obstacles, though, a special mention goes to the animation of the bubbles - on which I ultimately had to give up - settling for a compromise between time expenditure and result.

## Future Developments

There are many things that could improve the project when considering future developments, for the sake of brevity only two main features will be discussed:

1. Although the DSP is already relatively developed, a good distortion application usually affects the various bands of the audio spectrum differently. Splitting the signal into different bands and apply different types of distortion to them is definitively on a to-do list for future development.
2. The lack of abstract base classes in the project does not honour the actual knowledge of the OOP paradigm used into the development of this project, refactor the *Distortion* class and its object differently could definitively make the project more scalable.

## Appendix