Exploring DISSCO: A Guide for Digital Sound Synthesis and Composition

Hanna Chen, Gauransh Tandon, Megan Coleman, Nidhi Dholaria, Han Jiang, Daksh Varshney

School of Music in the College of Fine and Applied Arts at the University of Illinois at Urbana-Champaign

Mentored by Dr. Sever Tipei

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

INTRODUCTION

Based on the concept that there are an infinite number of possibilities, DISSCO is a sound synthesis and composition software. Learning DISSCO can be challenging, especially since its users come from a wide variety of backgrounds.

SOLUTION: In order to solve this problem, a manual and tutorial were created. These pieces enable users from all backgrounds to understand and implement the various components of DISSCO in creating entire compositions.

- THE MANUAL: These sections are intended to introduce the user to not only the definitions, but also the implementation, the software restrictions, and various analogies to ensure maximum comprehension.
- THE TUTORIAL: Enables the user with knowledge of the various components to manipulate the software and create their own unique projects.
- PROGRAMMING PARTIALS: Modified C++ code to store objects on the heap to preserve memory.

ENVELOPE LIBRARY

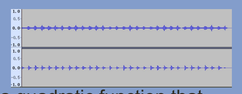
ENVELOPE: An envelope is an embedded function which can be implemented to represent the relationship between two variables. Each envelope in the envelope library outlines the behavior of the different features of sounds.

ENVELOPE LIBRARY: Functions are stored in the envelope library. A project can have several envelopes. Nodes are breakpoints that split the function into several pieces. An envelope can have several nodes (x>=2); more nodes lead to more complex songs. The x value of first node must be 0 and the x value of the last node must be 1, or the created piece of music will be clipped.

EXPERIMENTS: Below describes two examples of non-linear functions.

1. An envelope built with a quadratic function that concaves up sounds very unnatural. The transition between pitches has been emphasized. The graph is 30 rhombus with prominent apexes connected together.

Figure 1. Envelope of a quadratic concave up.



2. An envelope built with a quadratic function that concaves down sounds more natural and closer to an instrument. The transition between pitches has been weakened. The graph is 30 circles connected together.

Figure 2. Envelope of a quadratic concave down.

EVENT CLASS

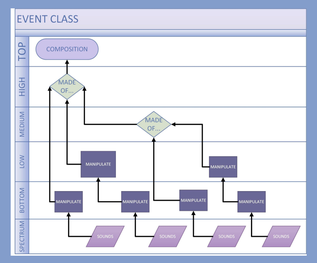


Figure 3. Event Class. As the main component of the composition module, the Event Class contains the Events at all levels for the project and defines their relationships.

The Structure's Purpose

The reasoning behind this hierarchical structure lies in its ability at each level to develop three key features:

- Randomness
- Variability
- Complexity

Enabling development upon the user-defined smallest sound, the Event Class introduces the capability of randomness at each level. Especially at higher levels which combine, alter, and manipulate those sounds.

From Code to Composition: Developing an Understanding

The Manual:

- Explanations targeting a wide audience
- Interconnections of the Parent-Child Relationship
- Analogies: Russian Dolls, C++, Family Trees

The Tutorial:

- Ten page document
- Explains direct connections between the interface and the resulting compositions
- User will develop a fundamental knowledge of the software.

FREQUENCY AND LOUDNESS

Loudness

Described in terms of sone units on a log scale.

Frequency & Frequency Modifiers

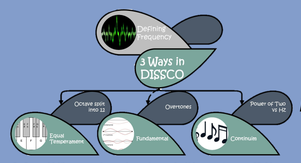


Figure 4: Defining Frequency. There are three ways to define frequency before modifiers are applied.

Modifiers include: Vibrato, Glissando, Bend, Tremolo

From Code to Composition: Developing an Understanding

The Manual:

- Implementation of method
- Discusses determining frequency & loudness

The Tutorial:

- Two page document
- Enables the user to connect the GUI with how the piece is synthesized

FUNCTIONS

As the primary way to introduce randomness in the software, functions are optional insertions available to the user. There are a number of different functions that can be used in the program, with varying levels of difficulty. The Functions section of the manual summarizes and explains the following functions, enabling users to manipulate the software with proficiency.

OVERVIEW OF FUNCTIONS

STOCHOS: A Function that works best for randomizing sound. It works by taking a total of three functions, two to set boundaries and one to pick values in between.

Figure 5. A Visual Representation of the Stochos Function.

It is important to note that this function requires a multiplier to work, as it works in decimals.

Figure 6 (right): An example of how the Stochos Function Would be set up in a Project.

In terms of composition, the stochos function is really good for having randomness between two very specific bounds. This makes it helpful for keeping the notes in the range of a specific instrument, for example.

RANDOM: Requires a minimum and maximum (double) bound and returns a decimal that is inclusive of the designated bounds.

RANDOMINT: Requires a minimum and maximum integer bounds and returns an integer that is inclusive of the designated bounds.

RANDOMIZER: Requires a (double) base value and a standard deviation (double) between 0.0 – 1.0. Returns a decimal that is within the range of base value plus or minus the base value times the standard deviation. For example, if the base value is 10 and the deviation is 0.5, the function would return a number between 5.0-15.0

SELECT: Allows a user to specify multiple possible values for an attribute. Ideal when values are not a part of a continuous range. A additional function must also be selected to specify how a value will chosen (randomly, by current child number, current type etc.).

The Manual:

- Explains in detail how the function works
- Tells the user about structural specifics that help ensure success

The Tutorial:

- Walks the user through a simple example for that function

PROGRAMMING PARTIALS

Working with files in C++ describing sounds based on frequency and amplitude, the software was modified to inculcate good software principles and phase shift was explored.

Inculcate Software Principles

In order to implement good software principles, two main issues were resolved:

- Extracting unrelated sub problems
- Preserving memory

Inculcated these principles in the following three files:

- partial.cpp
  - Takes the number of samples needed & associated duration
  - Sets the frequency and amplitude of these sounds accordingly
- loudness.cpp
  - Uses the partial file for calculating the loudness of the sound
- sound.cpp
  - Uses the loudness as just one component

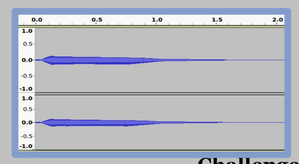


Figure 7. This sound has 1 partial whose frequency is unchanged but whose amplitude decreases after the first half.

Challenges

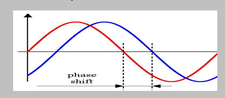
The main challenge was to preserve memory without hindering the performance simultaneously. The solution lied in the allocation of memory (initially run-time stack).

- Created objects on the heap
  - Increased speed and efficiency
  - Eased the extraction of objects as unrelated sub problems of the main partial file
- Preserved memory deletes objects on the heap
  - Avoids segmentation faults & memory leaks

Phase Shift

As unexplored territory, the phase shift is originally set to 0 throughout the software. With further exploration, this phase shift can be incorporated in the partial file.

Figure 8. Phase Shift as described by the equation:
$S = A \sin(2\pi(ft+\phi))$



CONCLUSION

The initial goal of better understanding the program from a user perspective & enabling users to manipulate the software in an informative way, was achieved. In addition, the software behind the scenes altered it's memory allocation to better service the programs needs. There are still many different features of the program that have been left unexplored and will require further work in the future. The manual and tutorial will serve as a strong foundation for further documentation.