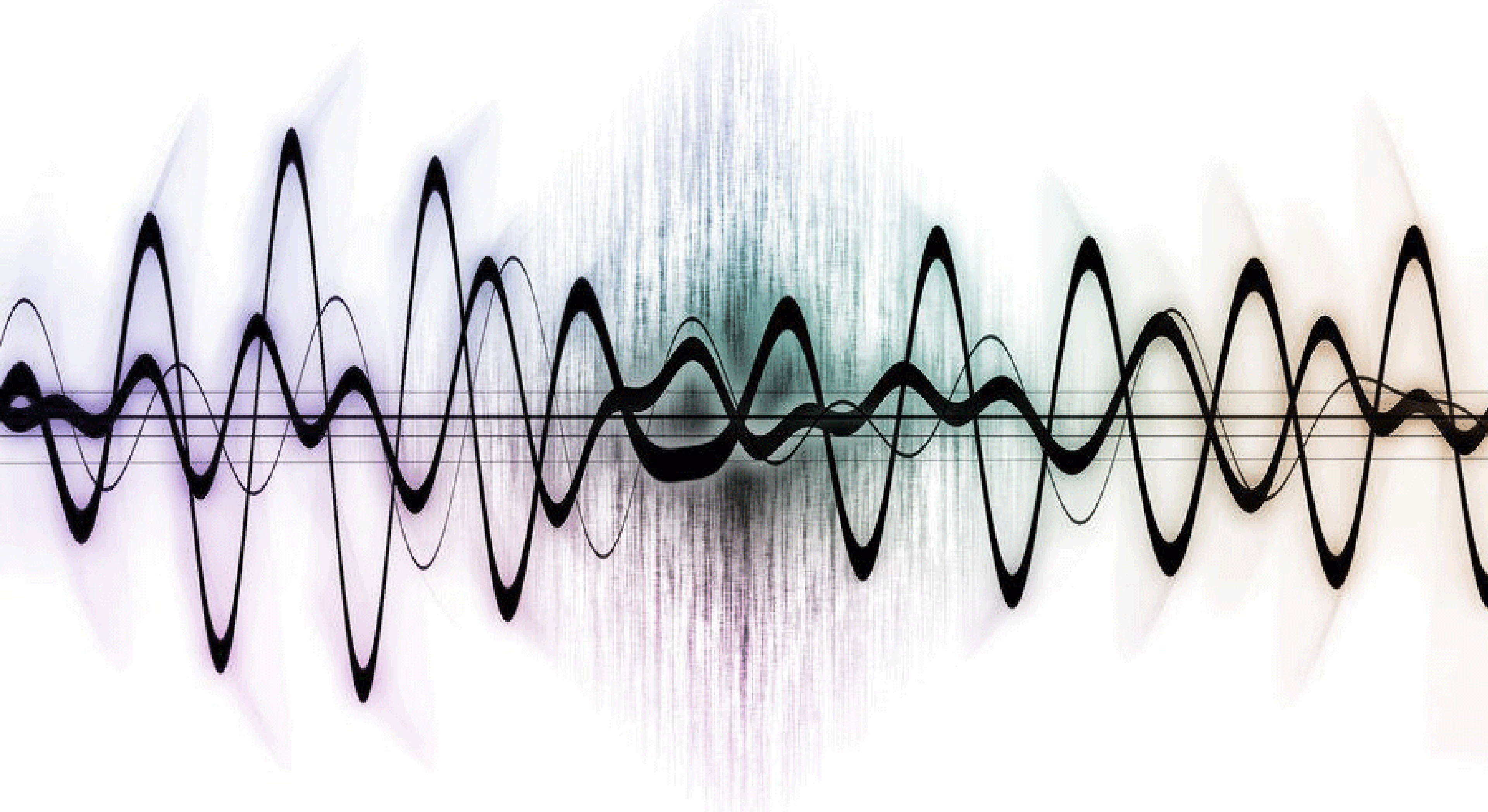

Tinkering Audio II: Further Notes on Digital Sound

Creative Computing – Michael Scott

Tinkering Audio 2

WORKSHOP ACTIVITIES

Base Tone Generator

```
OUTPUT_FILENAME = "noise.wav"
LENGTH_OF_FILE_IN_SECONDS = 1
CHANNEL_COUNT = 1
SAMPLE_WIDTH = 2                                                        # 2 bytes per sample
SAMPLE_RATE = 44100
SAMPLE_LENGTH = SAMPLE_RATE * LENGTH_OF_FILE_IN_SECONDS
COMPRESSION_TYPE = 'NONE'
COMPRESSION_NAME = 'not compressed'
MAX_VALUE = 32767
FREQUENCY = 15000

import wave
import struct
import math

noise_out = wave.open(OUTPUT_FILENAME, 'w')
noise_out.setparams((CHANNEL_COUNT, SAMPLE_WIDTH, SAMPLE_RATE,
    SAMPLE_LENGTH, 'NONE', 'not compressed'))

values = []

for i in range(0, SAMPLE_LENGTH):
    value = math.sin(2.0 * math.pi * FREQUENCY * (float(i) / SAMPLE_RATE)) \
        * MAX_VALUE
    packed_value = struct.pack('h', int(value))

    for j in range(0, CHANNEL_COUNT):
        values.append(packed_value)


noise_out.writeframes(b''.join(values))
noise_out.close()
```

Activity #1: Tone Generator

- If you have not yet implemented the tone generator, do so.
- Then, refactor the tone generator in order to:
  - Include arguments which:
    - Specify the frequency of the wave generated
    - Specify the amplitude of the wave generated
    - Specify the length (in seconds) of the tone generated
  - Decouple file input/output from wave calculations
  - Decouple the computation (i.e., replacing the sin() calculation on line 23 with a function call that returns a value)

Example

```
OUTPUT_FILENAME = "noise.wav"
LENGTH_OF_FILE_IN_SECONDS = 1
CHANNEL_COUNT = 1
SAMPLE_WIDTH = 2 # 2 bytes per sample
SAMPLE_RATE = 44100
SAMPLE_LENGTH = SAMPLE_RATE * LENGTH_OF_FILE_IN_SECONDS
COMPRESSION_TYPE = 'NONE'
COMPRESSION_NAME = 'not compressed'
MAX_VALUE = 32767
FREQUENCY = 4000

noise_out = wave.open(OUTPUT_FILENAME, 'w')

noise_out.setparams((CHANNEL_COUNT, SAMPLE_WIDTH, SAMPLE_RATE,
    SAMPLE_LENGTH, 'NONE', 'not compressed'))

noise_out.writeframes(
    package(generate_tone(440, 0.5))
)

noise_out.close()
```

Example

```
generate_tone(frequency, amplitude):

    values = []

    for i in range(0, SAMPLE_LENGTH):
        value = sin_wave(i, FREQUENCY, 1.0)
        values.append(value)

    return values


sin_wave(position, frequency, amplitude):

    return math.sin(2 * PI * frequency * (position / SAMPLE_RATE))
        * MAX_VALUE * amplitude
```

Example

```
package(tone_list):

    values = []

    for i in range(0, len(tone_list)):
        packed_value = struct.pack('h', int(tone_list[i]))
        values.append(packed_value)

    return b''.join(values)
```

Activity #2: Tone Combination

- Research multiple arguments and the continue keyword in Python
- Implement a function that combines tones together through addition
- Produce a wav files that combines the following pure tones:
  - 440 Hz
  - 880 Hz
  - 1320 Hz
  - 1760 Hz
- Function should be able to specify the frequency and amplitude of each wave
  - Compare uniform amplitude (where, the amplitude for each wave is the same) to decreasing amplitude (where, the amplitude for higher frequencies is halved each time)

Example

```
def combine_tones(*tones):

    max_length = 0

    for tone in list(tones):
        if len(tone) > max_length:
            max_length = len(tone)

    values = []

    for i in range(0, max_length):

        value = 0
        for tone in list(tones):
            if i >= len(tone):
                continue
            else:
                value += tone[i]

        if value > MAX_VALUE:
            values.append(MAX_VALUE)
        elif value < -MAX_VALUE:
            values.append(-MAX_VALUE)
        else:
            values.append(value)

    return values
```

Activity #3: Wave Shapes

- Define functions to calculate different types of wave:
  - Triangle Wave
  - Square Wave
  - Sawtooth Wave
- Use additive synthesis where appropriate, but also investigate how other computations (e.g., from fxSolver) could be leveraged to improve run-time performance

Example

```
convert_to_simple_square_wave(position, frequency, amplitude):

    base_value = sin(2 * PI * frequency * (position) / SAMPLE_RATE)
        * MAX_VALUE * amplitude

    if base_value < (-MAX_VALUE / 2):
        return -MAX_VALUE
    elif base_value > (MAX_VALUE / 2):
        return MAX_VALUE
    else:
        return 0
```

Activity #4: Harmonics

- Review the video at https://www.youtube.com/watch?v=YsZKvLnf7wU&
- Implement an algorithm that calculates a set of harmonics and outputs the when given the:
  - wave shape
  - fundamental frequency
  - fundamental amplitude
  - number of overtones
  - Length (in seconds) of the tone generated

Activity #5: Splice and Dice

- Implement an algorithm that will combine different tones that have been generated into a single wav file
- Produce a 10-second wav file which:
  - Plays a melody from 20 notes
  - For details, see: http://pages.mtu.edu/~suits/notefreqs.html

Example

```
>>> print range(1,3)
[1, 2]
>>> print range(3,1)
[]
>>> print range(-1,5)
[-1, 0, 1, 2, 3, 4]
>>> print range(1,100)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... 99]
```

Example

```
def increaseVolumeByRange(sound):
  for sampleNumber in range(0, getLength(sound)):
    value = getSampleValueAt(sound, sampleNumber)
    setSampleValueAt(sound, sampleNumber, value * 2)
```

```
def increaseVolume(sound):
  for sample in getSamples(sound):
    value = getSample(sample)
    setSample(sample,value * 2)
```

What is the difference between these two functions?

Example

```
def increaseAndDecrease(sound):
  length = getLength(sound)
 for index in range(0, length/2):
    value = getSampleValueAt(sound, index)
    setSampleValueAt(sound, index, value*2)
 for sampleIndex in range(length/2, length):
    value = getSampleValueAt(sound, index)
    setSampleValueAt(sound, index, value*0.2)
```