# Programming for Visual Artists

Aalto University
School of Arts, Design and Architecture

2024/2025
Department of Art and Media

# Some things to check

Processing reference:
https://processing.org/reference

# Recap

IN CASE YOU MISSED IT

- Processing and p5.js
- HTML and p5.js
- Image in Processing
- Video in Processing

**Welcome**

- Recap
- Today's Goals

**Webcam**

**Audio**

**Mixing Audio and Video**

**Examples**

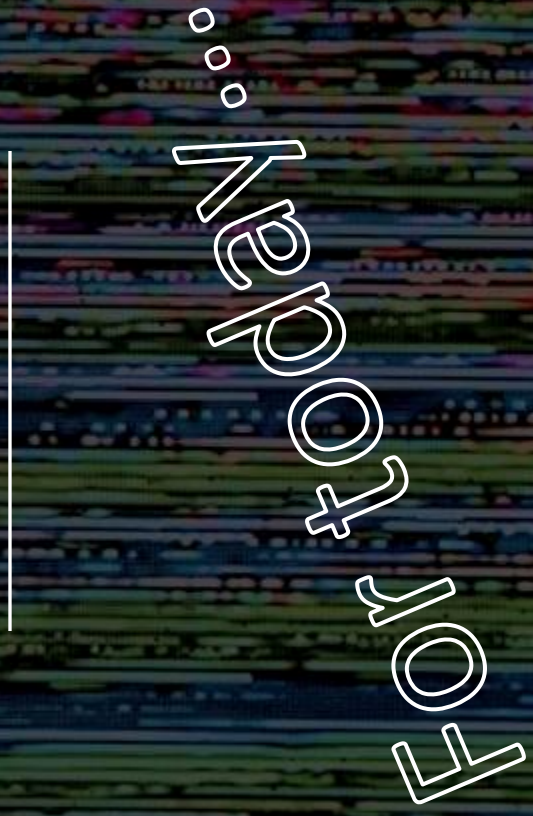
**BREAK (10:30–10:45)**

**Coding tasks**

- 3 bursts of particles
- Particles & Perlin Noise

**Q&A**

- For tomorrow…



# Processing - webcam

In Processing, webcam functionality is handled using the processing.video library, specifically the Capture class. This allows you to access and manipulate live video input from a connected webcam.

The processing.video library must be imported to use webcam features.

The Capture object is declared globally to handle the video stream.

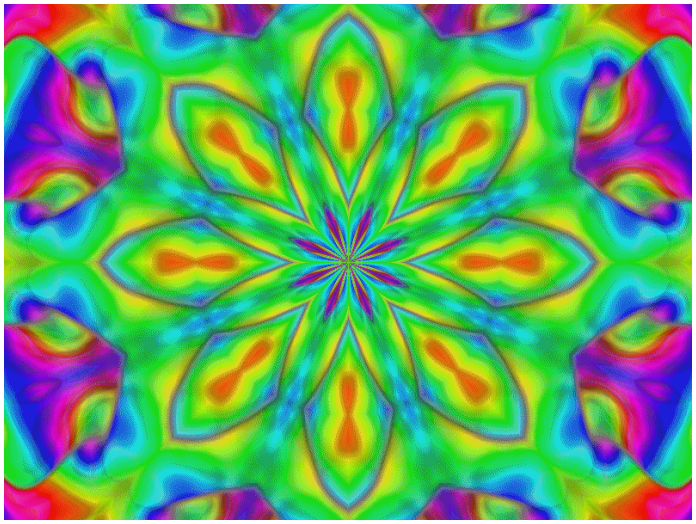
The Capture object is instantiated with the current sketch as a reference and the desired resolution. webcam.start(); is required to begin capturing video frames.

webcam.available() checks if a new frame is ready. webcam.read(); updates the frame.

```
import processing.video.*;

Capture webcam;

void setup() {
  size(640, 480);
  webcam = new Capture(this, 640, 480); // Initialize webcam with 640x480 resolution
  webcam.start(); // Start capturing
}

void draw() {
  if (webcam.available()) { // Check if a new frame is available
    webcam.read(); // Read the new frame
  }
  image(webcam, 0, 0); // Display the webcam feed
}
```

# Processing - webcam

Check extras here:
https://github.com/ptiagomp/aalto-programming-visual-artists-24-25/tree/main/Session-08_18032025



# Processing - Sound

The Sound library for Processing provides a simple way to work with audio. It can play, analyse, and synthesize sound.

It provides a collection of oscillators for basic wave forms, a variety of noise generators, and effects and filters to play and alter sound files and other generated sounds.

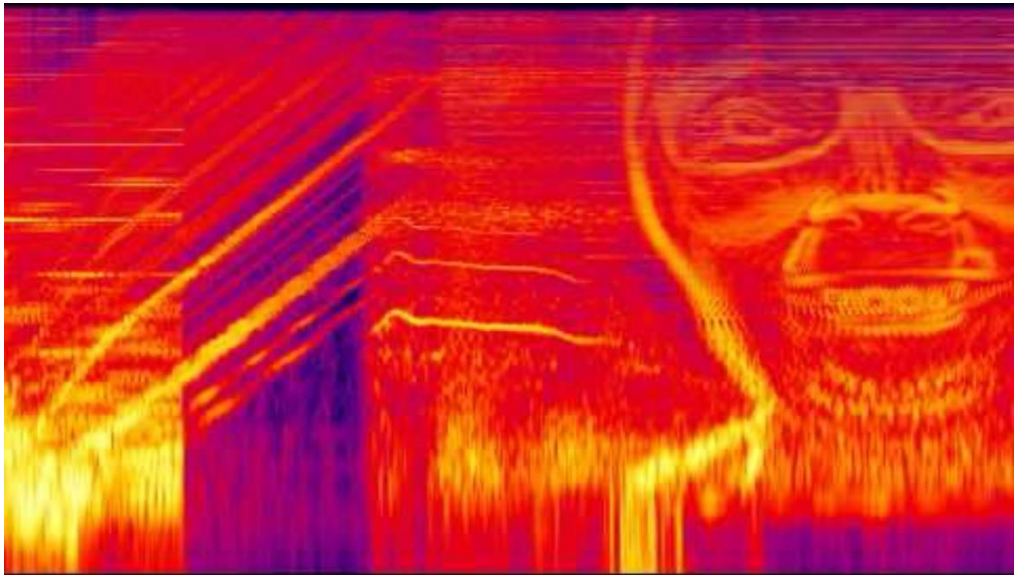
The syntax is minimal to make it easy to patch one sound object into another. The library also comes with example sketches covering many use cases to help you get started.

Grab the example sample from course's GitHub.

```
// Import the built-in Processing Sound library
import processing.sound.*;

SoundFile sound;  // Create a SoundFile object

void setup() {
  size(400, 200); // Set the window size

  // Load an audio file (place the file in the "data" folder)
  sound = new SoundFile(this, "sample.mp3");

  // Start playing the sound
  sound.play();
}

void draw() {
  background(0);  // Set background color to black
  fill(255);      // Set text color to white
  textSize(16);

  // Display instructions
  text("Press 'P' to pause, 'R' to restart", 50, 100);
}

// Handle keyboard input for controlling playback
void keyPressed() {
  if (key == 'p') {
    sound.stop();  // Stop playback (pause function not available)
  } else if (key == 'r') {
    sound.play();  // Restart the sound from the beginning
  }
}
```

# Processing - Audio

Image to audio – Aphex Twin example



Converting images into sound, like **Aphex Twin's hidden spectrogram images**, involves transforming visual data into an **audio spectrogram** using Fast Fourier Transform (FFT) analysis.

A spectrogram represents **time (X-axis), frequency (Y-axis), and amplitude (brightness or colour)**. Images can be **converted into frequencies**, were pixel intensity maps to loudness.

Each **column of pixels** in an image is treated as a frequency spectrum. The **brighter the pixel, the louder the frequency** at that point in time.

He likely used **Metasynth**, which allows importing an image as a spectrogram and generating audio. When played through an **FFT-based analyzer**, the hidden images appear in the spectrogram.

# Processing – Audio

While Processing itself doesn't directly support spectrogram audio generation, here's a **basic idea** of how you could **convert pixel brightness into sound frequencies** using the **Sound library.**

Check this example:

```
import processing.sound.*;

SoundFile sound;
SinOsc osc;  // Sine wave oscillator

void setup() {
  size(400, 400);

  // Load an image (grayscale recommended)
  PImage img = loadImage("image.png");
  img.loadPixels();

  osc = new SinOsc(this);
  osc.amp(0.5);  // Set amplitude
  osc.play();

  // Convert pixels into sound frequency
  int x = width / 2;  // Pick a column (center of image)
  for (int y = 0; y < img.height; y++) {
    float brightness = brightness(img.pixels[y * img.width + x]);
    float freq = map(brightness, 0, 255, 100, 2000);  // Map pixel brightness to freq
    osc.freq(freq);
    delay(50);  // Short delay between frequency changes
  }
}
```

# Processing - Audio

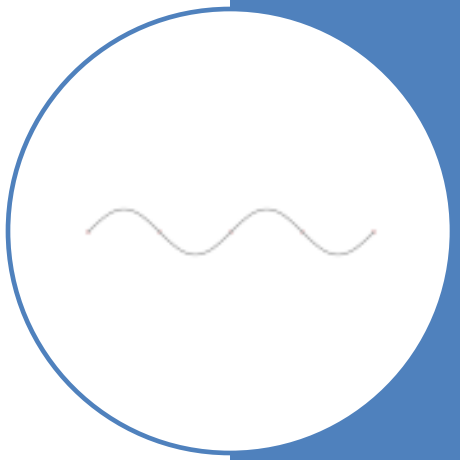
Processing's Sound library provides a variety of effects that can be applied to audio in real-time. These effects can modify sound in creative ways for music, interactive installations, and generative art.

Here's a list of some built-in Processing Sound library effects, their descriptions, and example use cases.

| Effect | Processing Class | Description | Use Cases |
|---|---|---|---|
| **Reverb** | `Reverb` | Simulates sound reflections for a sense of space. | Adding ambiance to speech, music, and environmental sounds. |
| **Delay** | `Delay` | Creates an echo effect by repeating the signal. | Used for rhythmic sound loops and spacey effects. |
| **Chorus** | `Chorus` | Detunes and delays sound for a richer tone. | Making vocals, synths, and instruments sound fuller. |
| **Flanger** | `Flanger` | Creates a sweeping, metallic modulation. | Adding motion to sound effects and experimental music. |
| **Filter (LowPass/HighPass/BandPass)** | `LowPass`, `HighPass`, `BandPass` | Controls specific frequency ranges of the sound. | Creating muffled effects, removing unwanted noise, or emphasizing parts of the spectrum. |
| **Distortion** | `Distortion` | Increases harmonics and adds grit to the sound. | Used in electric guitars, aggressive vocals, and glitch effects. |
| **Bitcrush (Lo-Fi Effect)** | `BitCrush` | Reduces sound resolution for a digital, noisy effect. | Used in retro-sounding music, 8-bit game sound effects. |
| **Envelope** | `Env` | Shapes the attack, decay, sustain, and release of sound. | Used in synthesizers and natural instrument modeling. |
| **Pan** | `Pan` | Adjusts the stereo position of sound (left-right). | Creates spatial effects, directional audio in games. |

# Processing – Audio + Video

The Fast Fourier Transform (FFT) is a mathematical algorithm that converts a signal from the time domain (waveform) to the frequency domain (spectrum). This is useful in audio processing, music visualization, and signal analysis because it helps us understand the individual frequency components of a sound.



**Raw Audio Data (Time Domain)**
An audio signal is a series of amplitude values over time.
When you visualize audio in software like Audacity or Processing's waveform, you see a waveform representation.

**Transforming to Frequency Domain**
The FFT algorithm takes this waveform and calculates how much of each frequency (e.g., bass, mid, treble) is present.
The result is a spectrum of frequencies and their intensities, which we can visualize as bars or waves.

**Logarithmic Scaling for Human Perception**
Humans perceive sound logarithmically, meaning we hear lower frequencies in more detail than higher ones.
This is why fft.logAverages(22, 3) is used in the Processing sketch—it groups frequency data into logarithmic bands for better visualization.

The function fft.logAverages(minBandwidth, bandsPerOctave) in Processing's Minim library is used to divide the frequency spectrum into logarithmically scaled bands instead of linear divisions. This makes the visualization more natural because it follows how human hearing perceives sound.

Check example in course's GitHub

# Processing – Audio + Video

**minBandwidth = 22**

This sets the **minimum frequency width** of the first band.

In this case, the first frequency band covers **22 Hz**.

Low frequencies (bass) need **smaller bands** because small differences in low frequencies are more perceptible to humans.

# Processing – Audio + Video

**bandsPerOctave = 3**

This defines **how many bands per octave** (octave = doubling of frequency, e.g., 100 Hz → 200 Hz).

3 means each **octave is divided into 3 bands**, leading to more precision in **low and mid frequencies** while still being efficient in high frequencies.

# Processing – Audio + Video

**Why Logarithmic Binning?**

A **linear frequency scale** (e.g., dividing from 0–22,050 Hz evenly) **doesn't match human hearing**. Humans hear **low frequencies with high precision** but **high frequencies with lower precision**.

**Example**:

Difference between **40 Hz and 60 Hz** is very noticeable.

Difference between **14,000 Hz and 14,200 Hz** is barely perceptible.

fft.logAverages(22, 3) helps **concentrate details in bass/mid frequencies** while simplifying high frequencies.

# Processing – Audio + Video or Image

Also, you can use audio to interact with your videos, images or webcam feed!

Processing makes it possible to manipulate images based on audio input, allowing you to create dynamic visual effects that respond to sound.

This is commonly used in visual music applications, sound-reactive art, and live performances.

**Some Techniques for Audio-Based Image Manipulation**

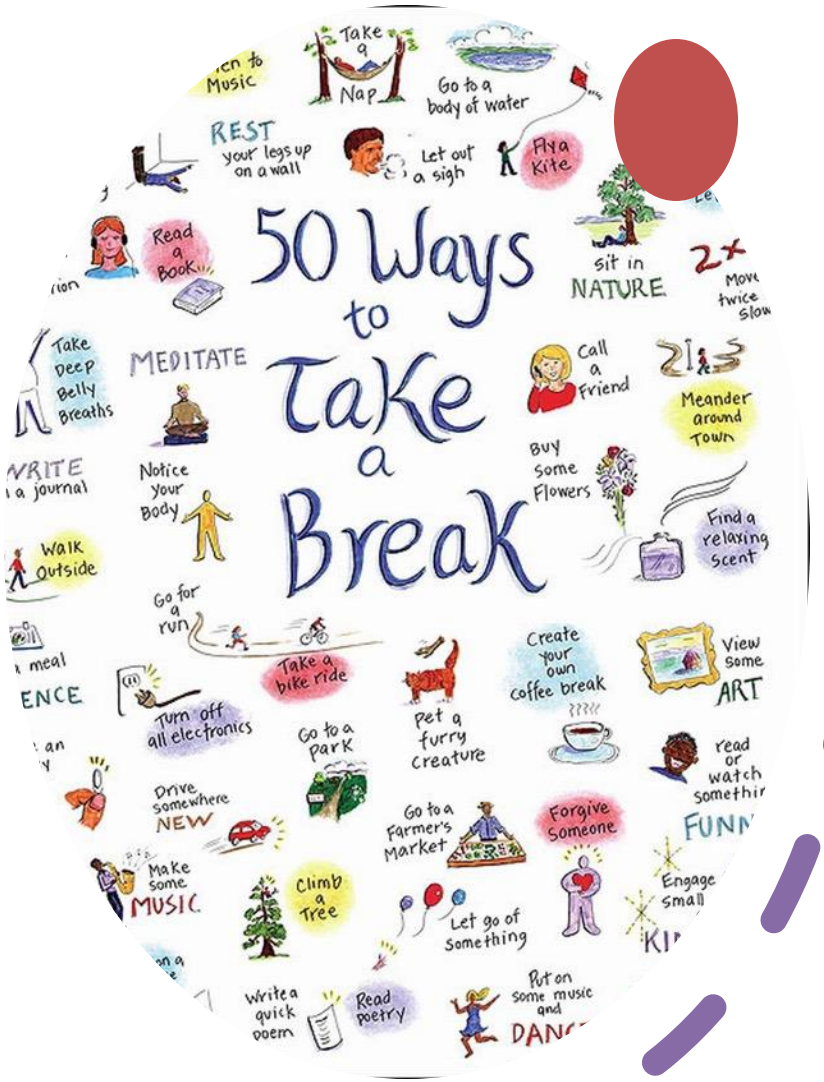
- Brightness & Transparency: Change pixel brightness based on audio volume.
- Distortion & Warping: Modify image position or distortion using frequency bands.
- Color Shifting: Adjust colors dynamically based on different audio frequencies.
- Pixelation & Blur Effects: Create pixelation or blur that reacts to volume.
- Waveform Visualizations: Use audio waveforms to generate dynamic effects.

Check examples on course's GitHub!

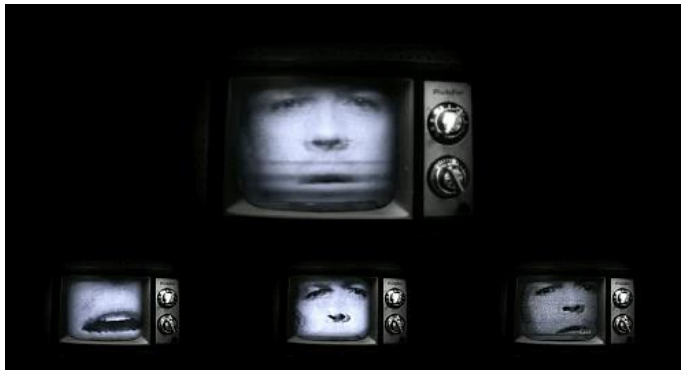
# Break

15 min.!

Please don’t be late!



# Hands-On Exercise!



**Objectives:**

- Change the codes provided.
- Explore Minim library

Get everything from here:
https://github.com/ptiagomp/aalto-programming-visual-artists-24-25/tree/main/Session-07_17032025

(if bored, check for the “extra” files!)


# Discussion & Q&A

**Share your feedback!**
Am I going too fast or too slow? Is this too easy or too hard?

**Next week's topics:**

- Fun extra stuff
- Project work!

**Don't forget the assignments!**