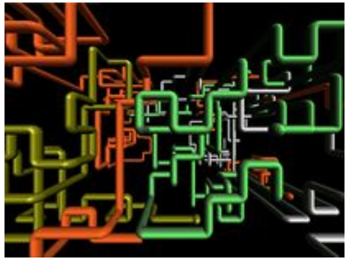
comp 371 project

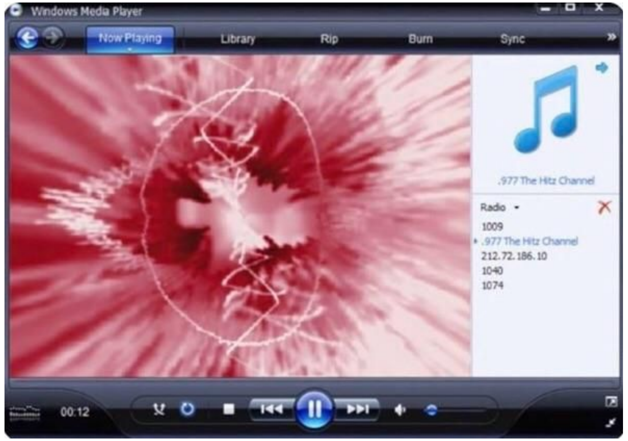
adrien trembley

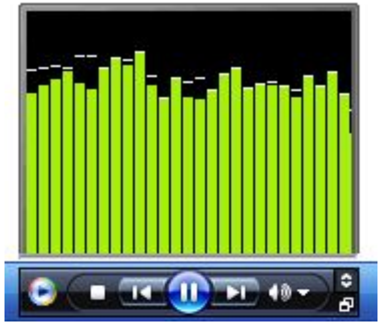


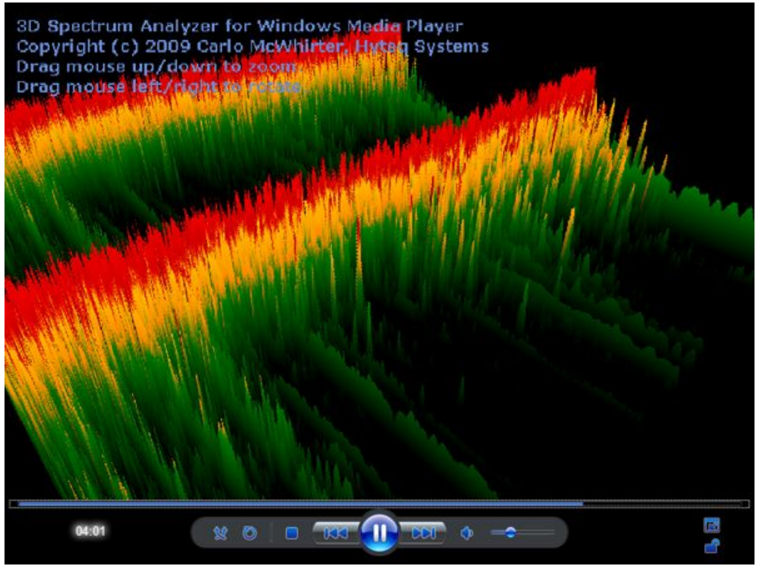
hi im adrien

me

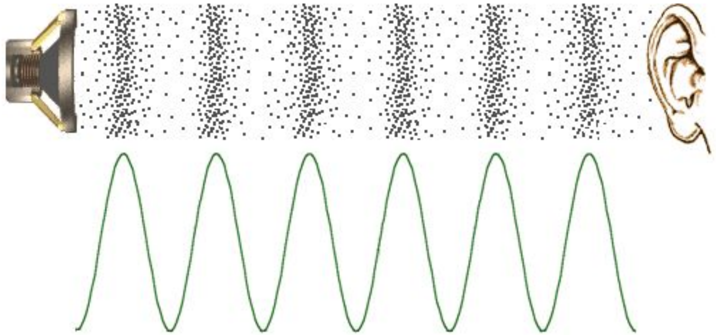
problem overview: Audio Visualization



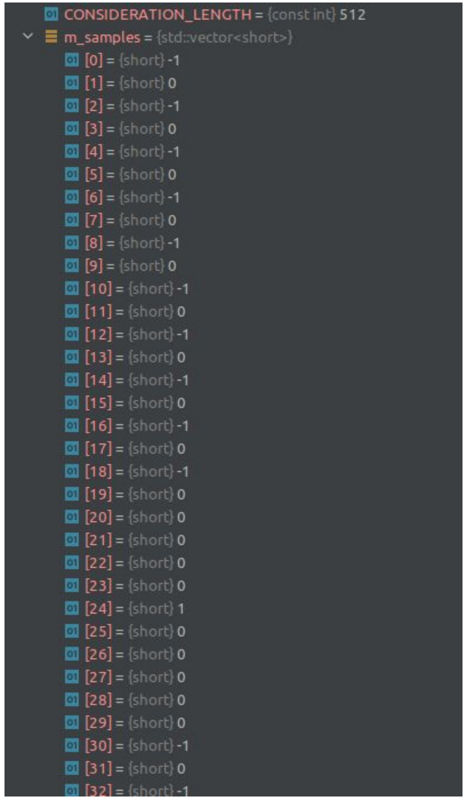



3D Spectrum Analyzer for Windows Media Player
Copyright (c) 2009 Carlo McWhirter, Hyteq Systems
Drag mouse up/down to zoom
Drag mouse left/right to rotate

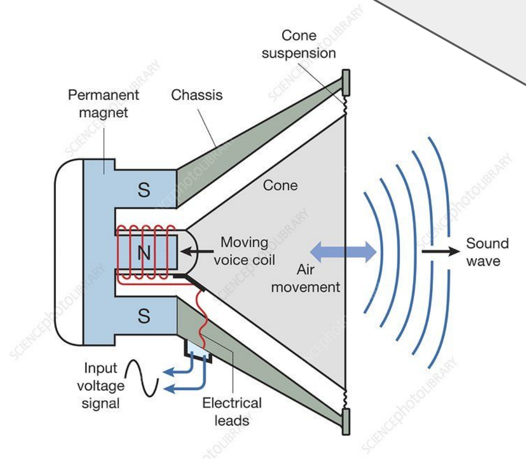
What the frick even is sound??



How do we store and play sound then???

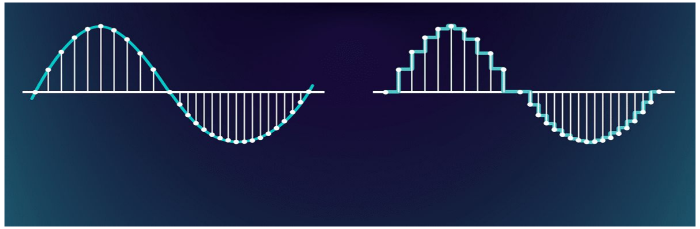


```
CONSIDERATION_LENGTH = {const int} 512
m_samples = {std::vector<short>}
  [0] = {short} -1
  [1] = {short} 0
  [2] = {short} -1
  [3] = {short} 0
  [4] = {short} -1
  [5] = {short} 0
  [6] = {short} -1
  [7] = {short} 0
  [8] = {short} -1
  [9] = {short} 0
  [10] = {short} -1
  [11] = {short} 0
  [12] = {short} -1
  [13] = {short} 0
  [14] = {short} -1
  [15] = {short} 0
  [16] = {short} -1
  [17] = {short} 0
  [18] = {short} -1
  [19] = {short} 0
  [20] = {short} 0
  [21] = {short} 0
  [22] = {short} 0
  [23] = {short} 0
  [24] = {short} 1
  [25] = {short} 0
  [26] = {short} 0
  [27] = {short} 0
  [28] = {short} 0
  [29] = {short} 0
  [30] = {short} -1
  [31] = {short} 0
  [32] = {short} -1
```

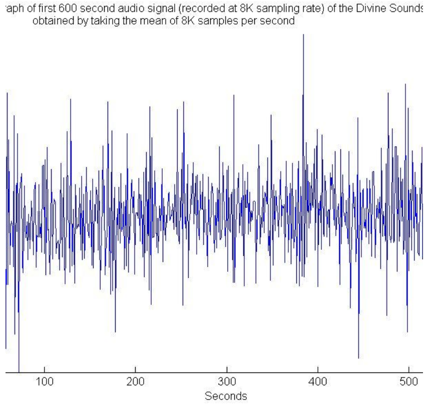


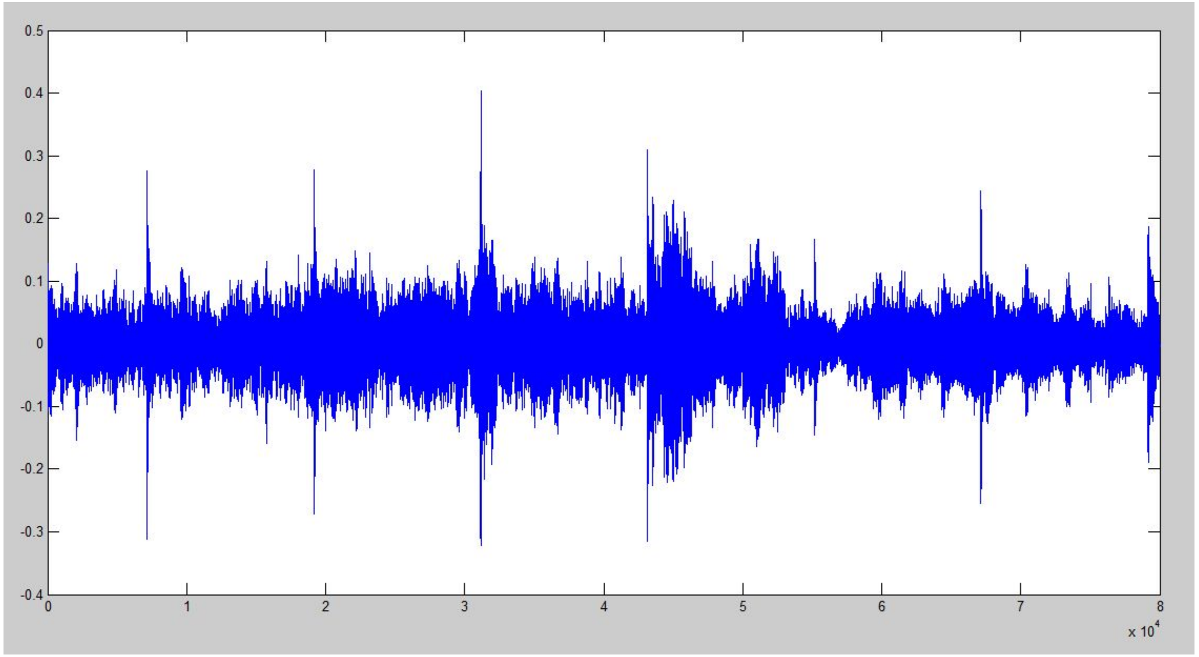
- Speaker position
- Two Channels (sometimes interleaved)

Analyzing the “samples”



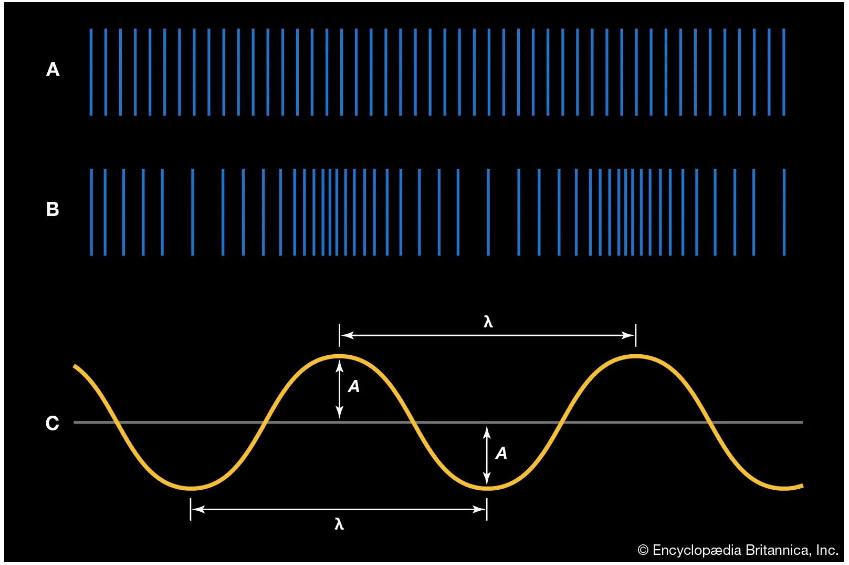
- This is a “Time domain” representation
- (each channel) is a complex wave



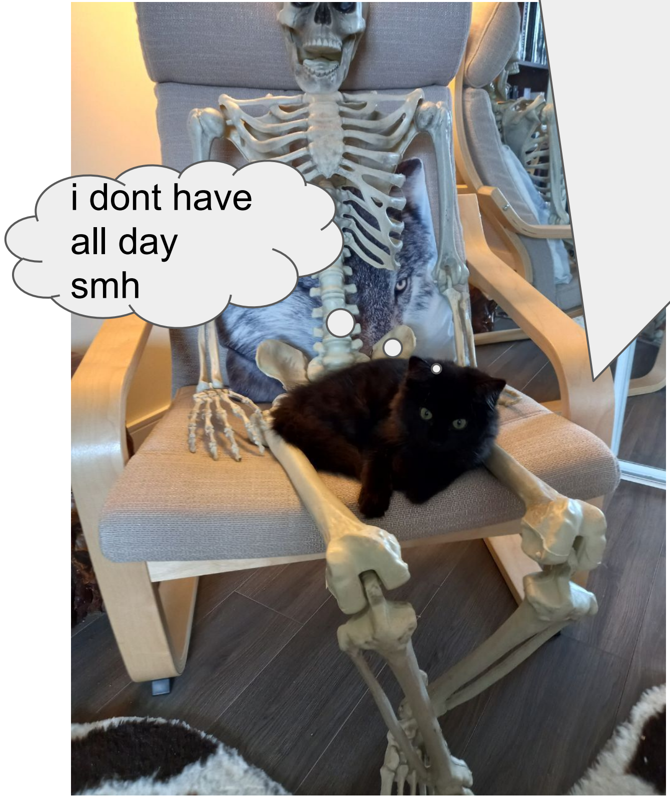


wait, How are multiple frequencies/notes stored in a single wave?

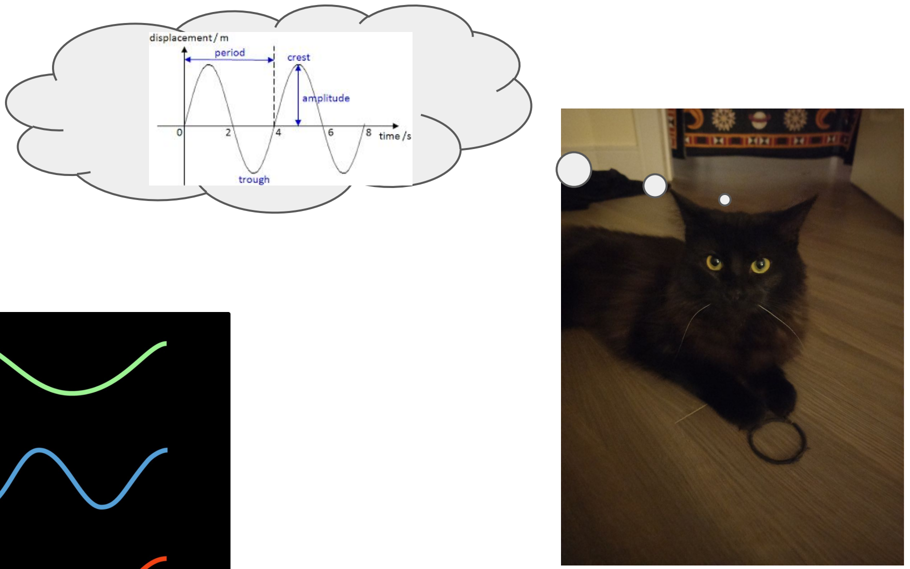
Answer: they are combined in the air to form complex compressions



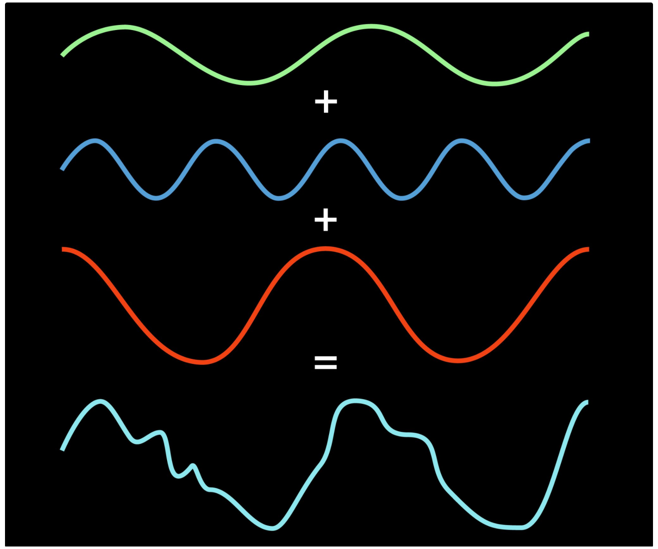
FINALLY, how do we find the frequencies and their amplitudes?

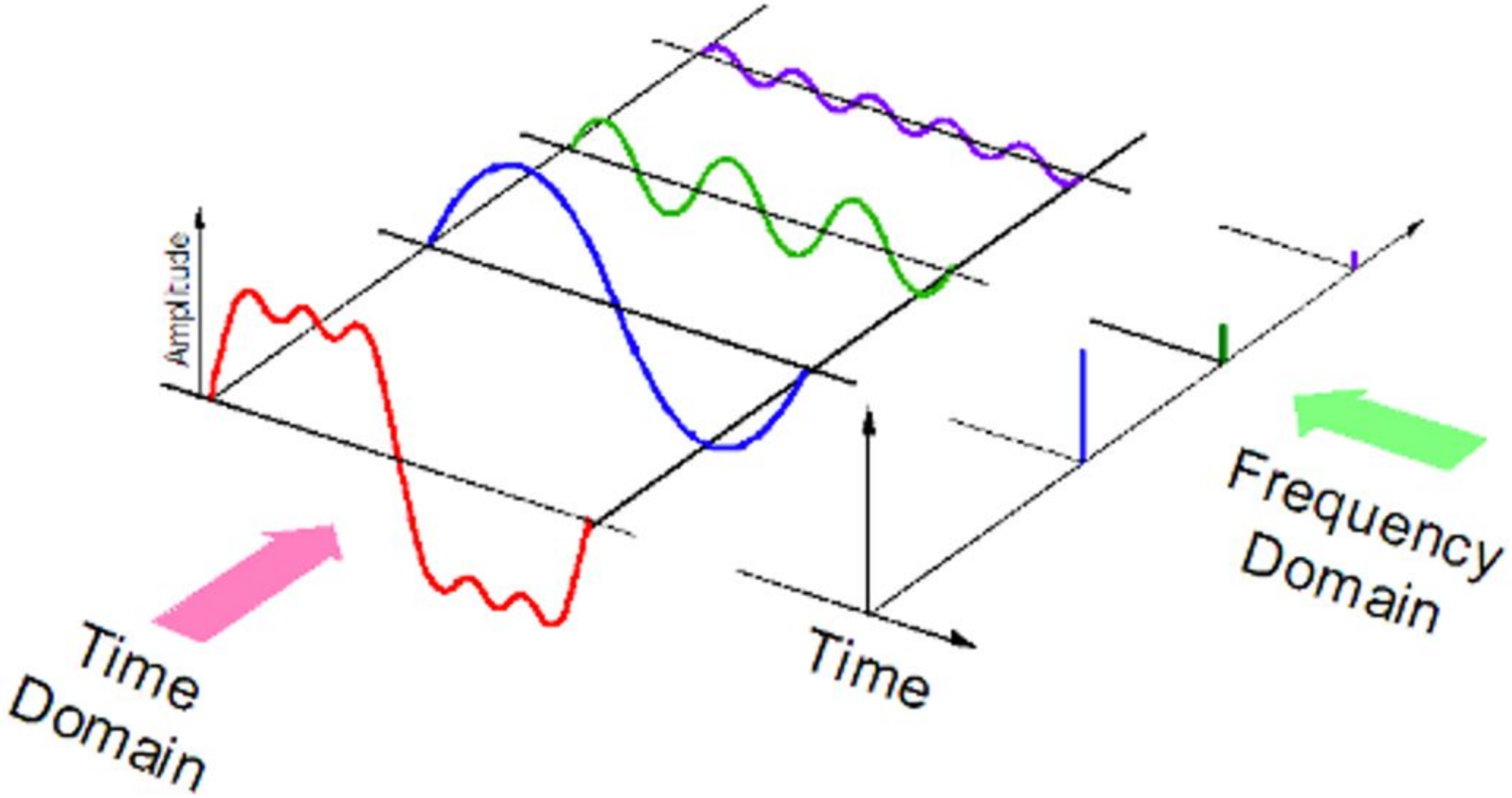


Lets think in waves



Σ

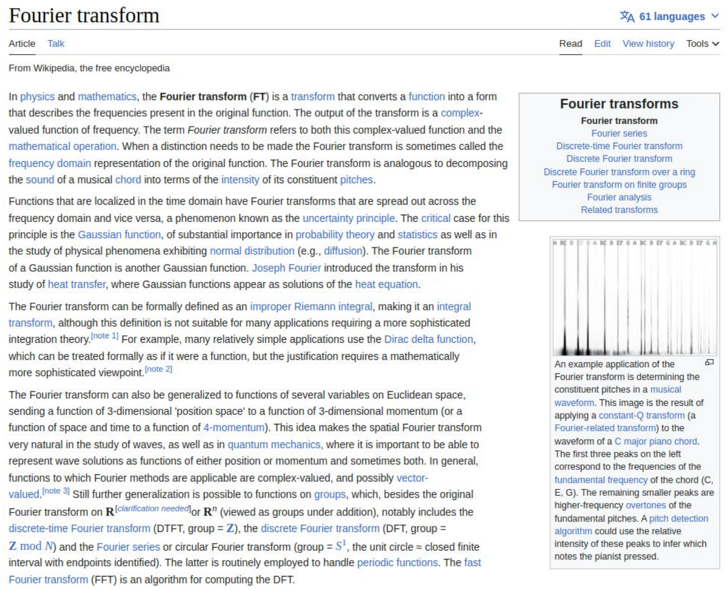



Amplitude
Time Domain
Time
Frequency Domain

How it’s done: complex math called the **Fourier Transform**

$$\int_{-\infty}^{\infty} F(k)\, e^{2\pi i k x}\, dk$$

Even I don’t really understand it fully. I won’t delve into the details



Fourier transform

61 languages

Article Talk

Read Edit View history Tools

From Wikipedia, the free encyclopedia

In physics and mathematics, the **Fourier transform** (**FT**) is a transform that converts a function into a form that describes the frequencies present in the original function. The output of the transform is a complex-valued function of frequency. The term *Fourier transform* refers to both this complex-valued function and the mathematical operation. When a distinction needs to be made the Fourier transform is sometimes called the frequency domain representation of the original function. The Fourier transform is analogous to decomposing the sound of a musical chord into terms of the intensity of its constituent pitches.

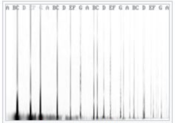
Functions that are localized in the time domain have Fourier transforms that are spread out across the frequency domain and vice versa, a phenomenon known as the uncertainty principle. The critical case for this principle is the Gaussian function, of substantial importance in probability theory and statistics as well as in the study of physical phenomena exhibiting normal distribution (e.g., diffusion). The Fourier transform of a Gaussian function is another Gaussian function. Joseph Fourier introduced the transform in his study of heat transfer, where Gaussian functions appear as solutions of the heat equation.

The Fourier transform can be formally defined as an improper Riemann integral, making it an integral transform, although this definition is not suitable for many applications requiring a more sophisticated integration theory.[note 1] For example, many relatively simple applications use the Dirac delta function, which can be treated formally as if it were a function, but the justification requires a mathematically more sophisticated viewpoint.[note 2]

The Fourier transform can also be generalized to functions of several variables on Euclidean space, sending a function of 3-dimensional 'position space' to a function of 3-dimensional momentum (or a function of space and time to a function of 4-momentum). This idea makes the spatial Fourier transform very natural in the study of waves, as well as in quantum mechanics, where it is important to be able to represent wave solutions as functions of either position or momentum and sometimes both. In general, functions to which Fourier methods are applicable are complex-valued, and possibly vector-valued.[note 3] Still further generalization is possible to functions on groups, which, besides the original Fourier transform on $\mathbf{R}$[*clarification needed*]or $\mathbf{R}^n$ (viewed as groups under addition), notably includes the discrete-time Fourier transform (DTFT, group = $\mathbf{Z}$), the discrete Fourier transform (DFT, group = $\mathbf{Z} \bmod N$) and the Fourier series or circular Fourier transform (group = S^1, the unit circle ≈ closed finite interval with endpoints identified). The latter is routinely employed to handle periodic functions. The fast Fourier transform (FFT) is an algorithm for computing the DFT.

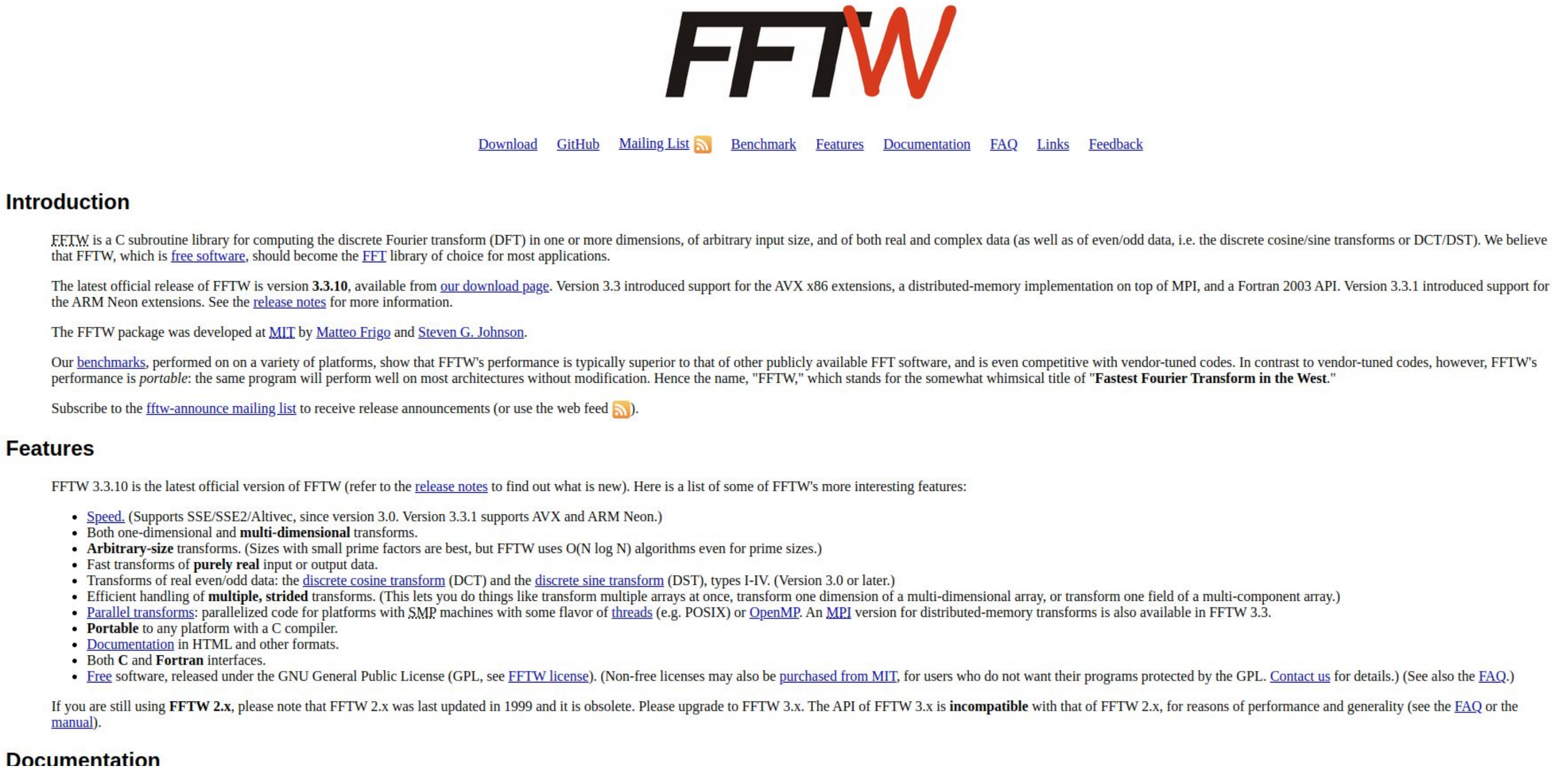
Fourier transforms

Fourier transform
Fourier series
Discrete-time Fourier transform
Discrete Fourier transform
Discrete Fourier transform over a ring
Fourier transform on finite groups
Fourier analysis
Related transforms



An example application of the Fourier transform is determining the constituent pitches in a musical waveform. This image is the result of applying a constant-Q transform (a Fourier-related transform) to the waveform of a C major piano chord. The first three peaks on the left correspond to the frequencies of the fundamental frequency of the chord (C, E, G). The remaining smaller peaks are higher-frequency overtones of the fundamental pitches. A pitch detection algorithm could use the relative intensity of these peaks to infer which notes the pianist pressed.

How it’s done *specifically* for this project: the **FFTW** c++ library



FFTW

Download GitHub Mailing List Benchmark Features Documentation FAQ Links Feedback

Introduction

FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data (as well as of even/odd data, i.e. the discrete cosine/sine transforms or DCT/DST). We believe that FFTW, which is free software, should become the FFT library of choice for most applications.

The latest official release of FFTW is version **3.3.10**, available from our download page. Version 3.3 introduced support for the AVX x86 extensions, a distributed-memory implementation on top of MPI, and a Fortran 2003 API. Version 3.3.1 introduced support for the ARM Neon extensions. See the release notes for more information.

The FFTW package was developed at MIT by Matteo Frigo and Steven G. Johnson.

Our benchmarks, performed on on a variety of platforms, show that FFTW's performance is typically superior to that of other publicly available FFT software, and is even competitive with vendor-tuned codes. In contrast to vendor-tuned codes, however, FFTW's performance is *portable*: the same program will perform well on most architectures without modification. Hence the name, "FFTW," which stands for the somewhat whimsical title of "**Fastest Fourier Transform in the West**."

Subscribe to the fftw-announce mailing list to receive release announcements (or use the web feed).

Features

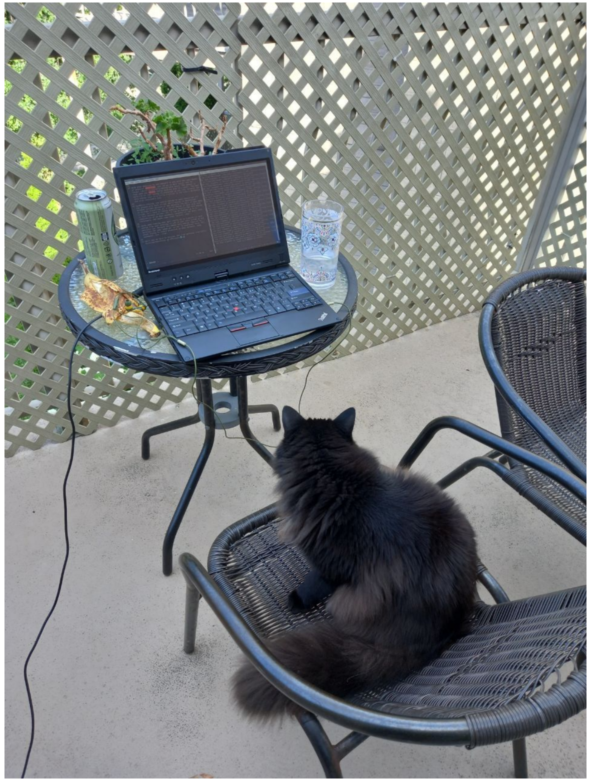
FFTW 3.3.10 is the latest official version of FFTW (refer to the release notes to find out what is new). Here is a list of some of FFTW's more interesting features:

- Speed. (Supports SSE/SSE2/Altivec, since version 3.0. Version 3.3.1 supports AVX and ARM Neon.)
- Both one-dimensional and **multi-dimensional** transforms.
- **Arbitrary-size** transforms. (Sizes with small prime factors are best, but FFTW uses O(N log N) algorithms even for prime sizes.)
- Fast transforms of **purely real** input or output data.
- Transforms of real even/odd data: the discrete cosine transform (DCT) and the discrete sine transform (DST), types I-IV. (Version 3.0 or later.)
- Efficient handling of **multiple, strided** transforms. (This lets you do things like transform multiple arrays at once, transform one dimension of a multi-dimensional array, or transform one field of a multi-component array.)
- Parallel transforms: parallelized code for platforms with SMP machines with some flavor of threads (e.g. POSIX) or OpenMP. An MPI version for distributed-memory transforms is also available in FFTW 3.3.
- **Portable** to any platform with a C compiler.
- Documentation in HTML and other formats.
- Both **C** and **Fortran** interfaces.
- Free software, released under the GNU General Public License (GPL, see FFTW license). (Non-free licenses may also be purchased from MIT, for users who do not want their programs protected by the GPL. Contact us for details.) (See also the FAQ.)

If you are still using **FFTW 2.x**, please note that FFTW 2.x was last updated in 1999 and it is obsolete. Please upgrade to FFTW 3.x. The API of FFTW 3.x is **incompatible** with that of FFTW 2.x, for reasons of performance and generality (see the FAQ or the manual).

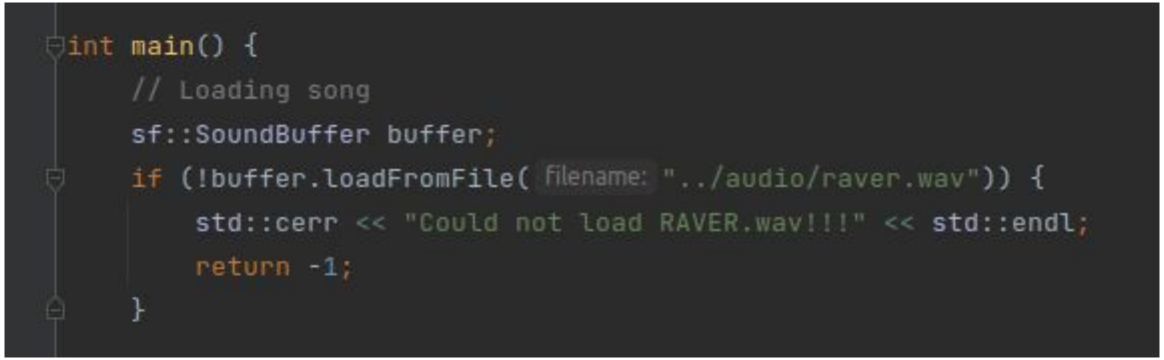
Documentation

Overview of my application structure



Step 1: Load the audio samples

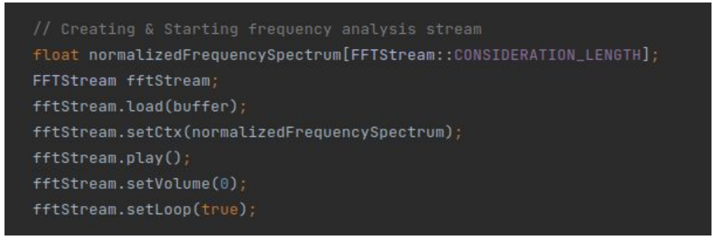
- Used the SFML-Audio library to do this
- Also used SFML to create my application window



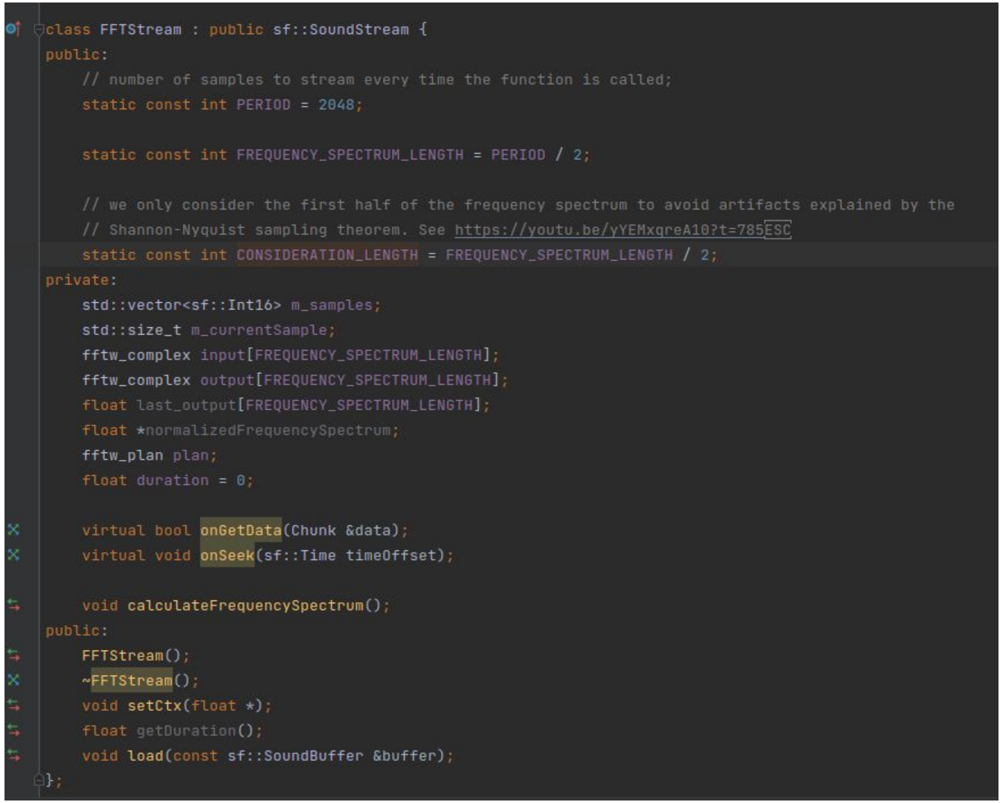
```
int main() {
    // Loading song
    sf::SoundBuffer buffer;
    if (!buffer.loadFromFile( Filename: "../audio/raver.wav")) {
        std::cerr << "Could not load RAVER.wav!!!" << std::endl;
        return -1;
    }
```

Step 2: Play audio & simultaneously use FFT to get frequency amplitudes

- Created a custom custom SoundStream class that runs the Fourier Transform while playing audio and stores the result in the *normalizedFrequencySpectrum* Float array



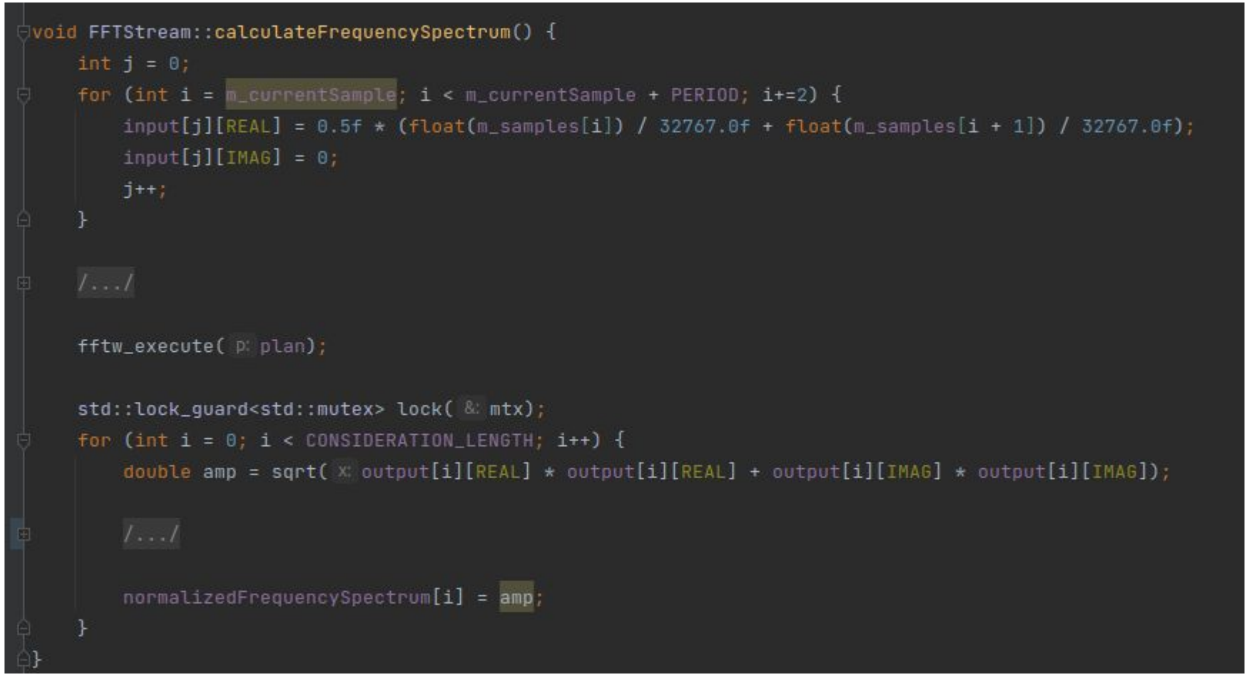
```
// Creating & Starting frequency analysis stream
float normalizedFrequencySpectrum[FFTStream::CONSIDERATION_LENGTH];
FFTStream fftStream;
fftStream.load(buffer);
fftStream.setCtx(normalizedFrequencySpectrum);
fftStream.play();
fftStream.setVolume(0);
fftStream.setLoop(true);
```



```
class FFTStream : public sf::SoundStream {
public:
    // number of samples to stream every time the function is called;
    static const int PERIOD = 2048;

    static const int FREQUENCY_SPECTRUM_LENGTH = PERIOD / 2;

    // we only consider the first half of the frequency spectrum to avoid artifacts explained by the
    // Shannon-Nyquist sampling theorem. See https://youtu.be/yYEMxqreA10?t=785ESC
    static const int CONSIDERATION_LENGTH = FREQUENCY_SPECTRUM_LENGTH / 2;
private:
    std::vector<sf::Int16> m_samples;
    std::size_t m_currentSample;
    fftw_complex input[FREQUENCY_SPECTRUM_LENGTH];
    fftw_complex output[FREQUENCY_SPECTRUM_LENGTH];
    float last_output[FREQUENCY_SPECTRUM_LENGTH];
    float *normalizedFrequencySpectrum;
    fftw_plan plan;
    float duration = 0;

    virtual bool onGetData(Chunk &data);
    virtual void onSeek(sf::Time timeOffset);

    void calculateFrequencySpectrum();
public:
    FFTStream();
    ~FFTStream();
    void setCtx(float *);
    float getDuration();
    void load(const sf::SoundBuffer &buffer);
};
```

Aside: The critical code that does the FFT

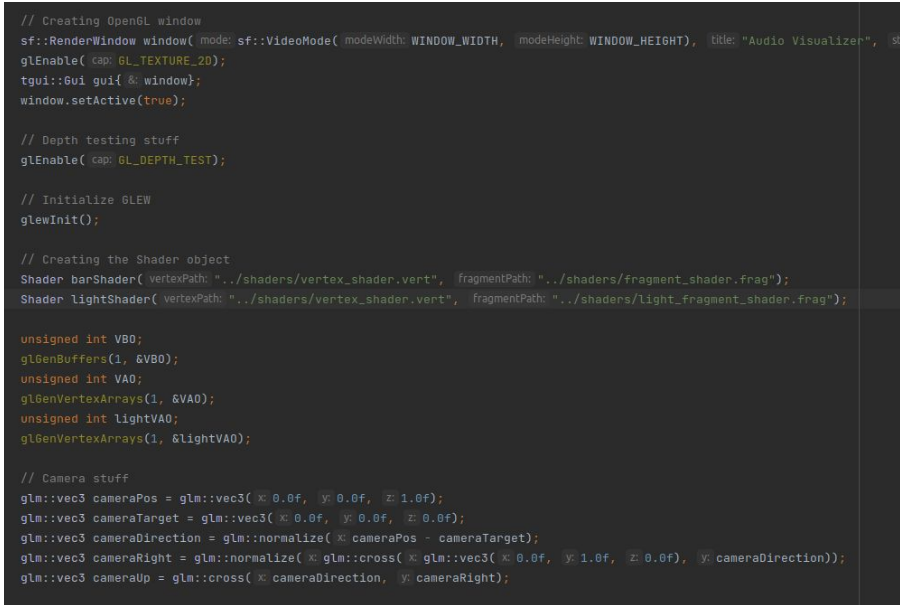
Could be improved to produce more accurate values, would need to do more research into wave analysis



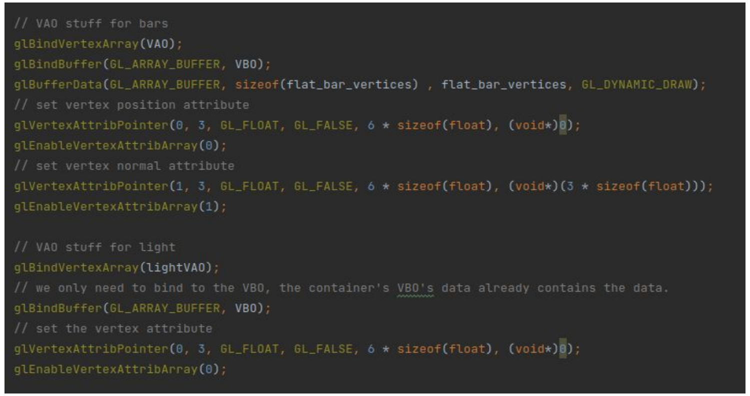
```
void FFTStream::calculateFrequencySpectrum() {
    int j = 0;
    for (int i = m_currentSample; i < m_currentSample + PERIOD; i+=2) {
        input[j][REAL] = 0.5f * (float(m_samples[i]) / 32767.0f + float(m_samples[i + 1]) / 32767.0f);
        input[j][IMAG] = 0;
        j++;
    }

    /.../

    fftw_execute( p: plan);

    std::lock_guard<std::mutex> lock( &: mtx);
    for (int i = 0; i < CONSIDERATION_LENGTH; i++) {
        double amp = sqrt( x: output[i][REAL] * output[i][REAL] + output[i][IMAG] * output[i][IMAG]);

        /.../

        normalizedFrequencySpectrum[i] = amp;
    }
}
```

Step 3: OpenGL stuff

- Create window
- Create VAO, VBOS (one for flat bars, one for 3D bars)
- Load and compile shaders
- Setup projection matrix
- Set up gui



```
// Creating OpenGL window
sf::RenderWindow window( mode: sf::VideoMode( modeWidth: WINDOW_WIDTH, modeHeight: WINDOW_HEIGHT), title: "Audio Visualizer", st
glEnable( cap: GL_TEXTURE_2D);
tgui::Gui gui{ &: window};
window.setActive(true);

// Depth testing stuff
glEnable( cap: GL_DEPTH_TEST);

// Initialize GLEW
glewInit();

// Creating the Shader object
Shader barShader( vertexPath: "../shaders/vertex_shader.vert", fragmentPath: "../shaders/fragment_shader.frag");
Shader lightShader( vertexPath: "../shaders/vertex_shader.vert", fragmentPath: "../shaders/light_fragment_shader.frag");

unsigned int VBO;
glGenBuffers(1, &VBO);
unsigned int VAO;
glGenVertexArrays(1, &VAO);
unsigned int lightVAO;
glGenVertexArrays(1, &lightVAO);

// Camera stuff
glm::vec3 cameraPos = glm::vec3( x: 0.0f, y: 0.0f, z: 1.0f);
glm::vec3 cameraTarget = glm::vec3( x: 0.0f, y: 0.0f, z: 0.0f);
glm::vec3 cameraDirection = glm::normalize( x: cameraPos - cameraTarget);
glm::vec3 cameraRight = glm::normalize( x: glm::cross( x: glm::vec3( x: 0.0f, y: 1.0f, z: 0.0f), y: cameraDirection));
glm::vec3 cameraUp = glm::cross( x: cameraDirection, y: cameraRight);
```



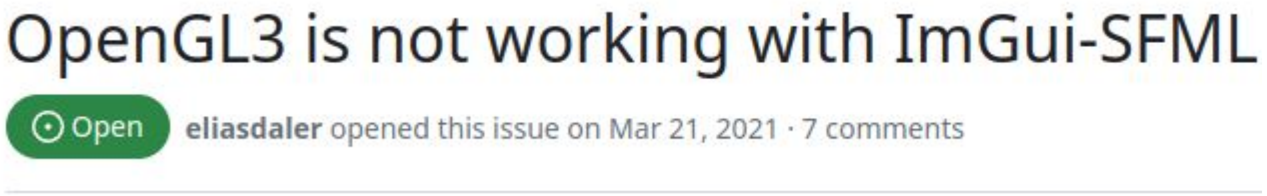
```
// VAO stuff for bars
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(flat_bar_vertices) , flat_bar_vertices, GL_DYNAMIC_DRAW);
// set vertex position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// set vertex normal attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);

// VAO stuff for light
glBindVertexArray(lightVAO);
// we only need to bind to the VBO, the container's VBO's data already contains the data.
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// set the vertex attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Step 4: Main program loop

```
While (running) {

        1.  Handle events (viewport size adjust + GUI)
        2.  Calculate height=amplitude for each bar (each bar represents a frequency range)
        3.  Calculate view matrix (camera)
        4.  Foreach (bar in bars) {
                     i.  Calculate bar model matrix
                    ii.  Draw the bar (using either the vertices for a flat or voluminous bar)

            }

        5.  Draw light source (optional)
        6.  Draw GUI

}
```

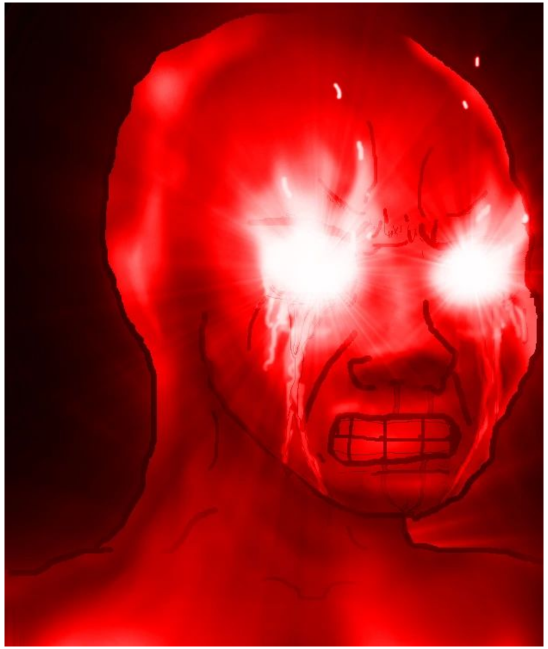
Note: Things like the view matrix, model matrix for bars, VBO used, etc… change depending on program variables which are modified using the GUI

Libraries Used

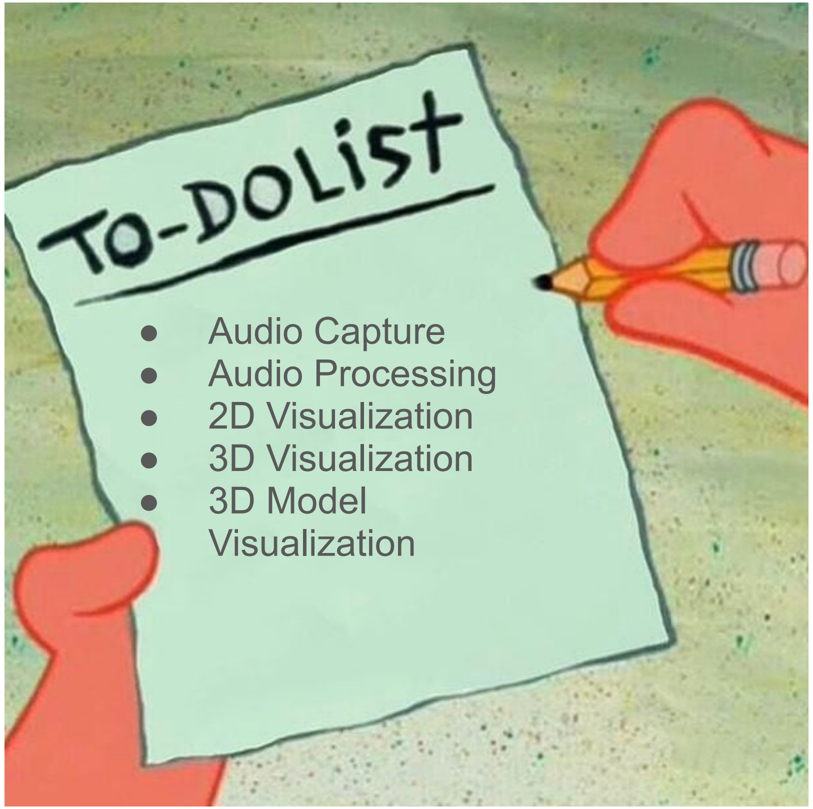

OpenGL3 is not working with ImGui-SFML

Open **eliasdaler** opened this issue on Mar 21, 2021 · 7 comments

- **SFML:** For windowing, basic audio functions
- **GLEW:** To help find and access OpenGL functionality
- **TGUI:** For the GUI (imGui + SFML + OpenGL3 doesn't mix well)
- **FFTW:** For the Fast Fourier Transform
- **Assimp:** Model Loading

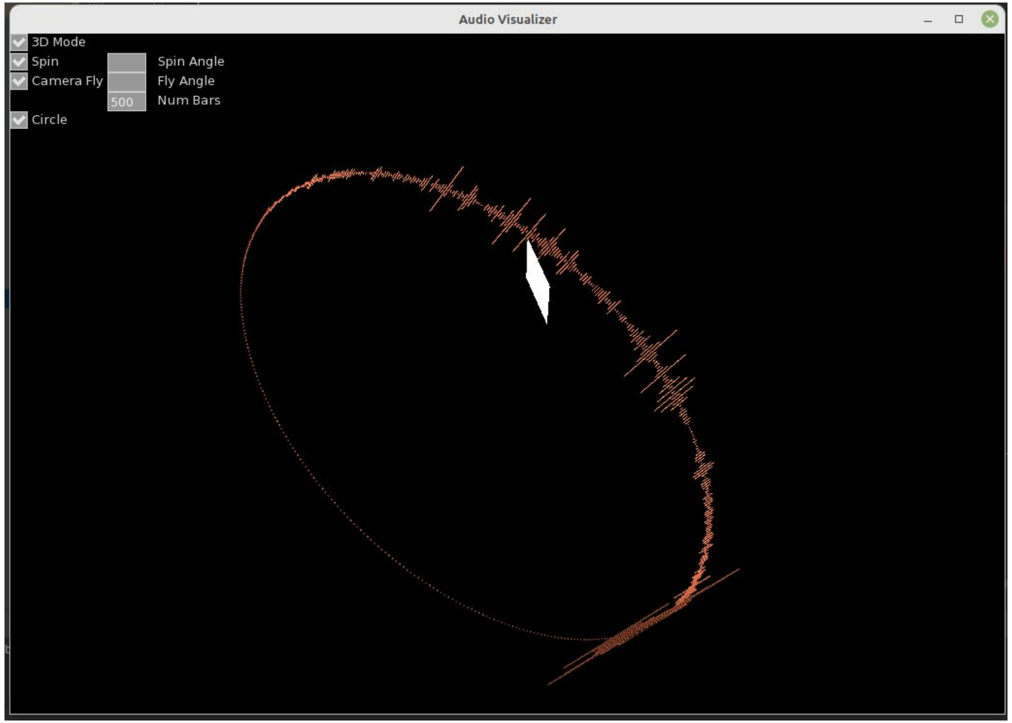


Goals outlined in my Project Proposal



- Audio Capture
- Audio Processing
- 2D Visualization
- 3D Visualization
- 3D Model Visualization

Demo time


Audio Visualizer

- ✓ 3D Mode
- ✓ Spin — Spin Angle
- ✓ Camera Fly — Fly Angle
- 500 — Num Bars
- ✓ Circle