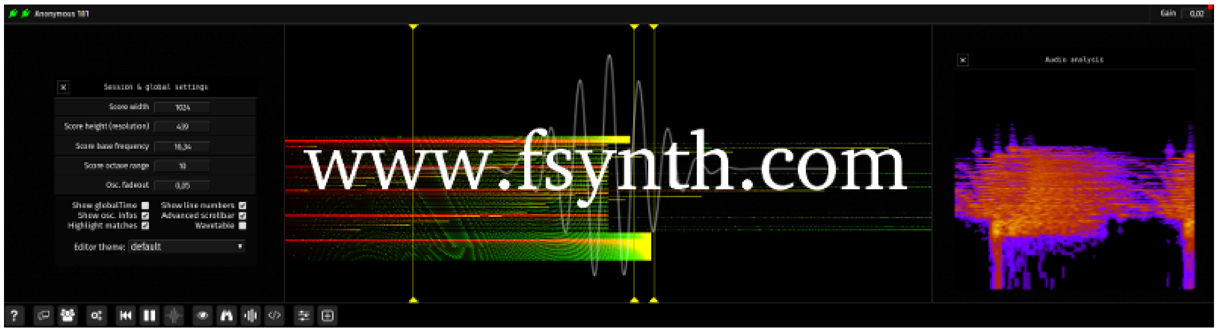
Fragment

The Collaborative Spectral Synthesizer

by Julien Verneuil - contact@fsynth.com

This is the documentation of the Fragment synthesizer.



Session & global settings

Score width 1024
Score height (resolution) 439
Score base frequency 16,34
Score octave range 10
Osc. fadeout 0,05
Show globalTime ☐ Show line numbers ☑
Show osc. infos ☑ Advanced scrollbar ☑
Highlight matches ☑ Wavetable ☐
Editor theme: default

(https://www.fsynth.com)

Table of Contents

Contents

1 Introduction

The **Fragment synthesizer** (`https://www.fsynth.com`) (also called fsynth) is a collaborative web-based musical instrument which allow direct manipulation of the sound spectrum by the use of on-the-fly GPU (Graphics Processing Unit) programming.

Fragment is stereophonic, polyphonic, multitimbral and support live coding of audio and visuals at the same time.

Fragment is first and foremost a powerfull additive synthesizer which let you have complete control over the sound spectrum in real-time with direct visual feedback.

Many videos of Fragment live sessions were recorded and are available on YouTube as a playlist (`https://www.youtube.com/playlist?list=PLYhyS2OKJmqe_PEimydWZN1KbvCzkjgeI`).

1.1 History

In 2009, i discovered the Virtual ANS (`http://www.warmplace.ru/soft/ans/`) synthesizer by Alexander Zolotov (`http://www.warmplace.ru`), a software emulation of the Russian photoelectronic synthesizer ANS (`http://en.wikipedia.org/wiki/ANS_synthesizer`) which was created by the Russian engineer Evgeny Murzin (`http://en.wikipedia.org/wiki/Evgeny_Murzin`) from 1938 to 1958.

I was blown away by the remarquable possibilities offered by the Virtual ANS which let you draw the sound spectrum over time, i then discovered programs such as MetaSynth or HighC (`https://highc.org`), this is how i started to experiment with the "drawn sound" method.

Fragment started in 2015, when i was making the first prototype of a web-based ANS-like synthesizer software which is still a work in progress, a prototype of Fragment was made in a single week, the prototype was quite mature but lacked in features, i considered Fragment as a pretty but failed experiment at the time.

In the summer 2016 while i was releasing the prototype source code, i played with it again and after tweaking the GLSL code, i was able to make some nice sounds and envision what would be possible with this technology, so i started to think about that prototype again and come up with many ideas that would make Fragment more functional and easier to use, the real work begun.

After many hours spent crafting the software, the first version of Fragment was released in January 2017, it was quite limited at that time, Fragment has now improved alot.

1.2 Capabilities

- Powerful additive synthesizer
  - Powered by WebAudio oscillators
  - Powered by FAS, an independent program
  - Powered by a wavetable
- Stereophonic
- Monaural
- Polyphonic
  - Automatically detected from the GPU capabilities
  - 16 notes minimum
  - 704 notes with a GeForce GTX 970 GPU
- Multitimbral
- Aliasing free
- Adjustable audio output channel per slices
  - Multiple audio channels are only supported with FAS
- Shader inputs, webcam, textures and more to come
- Real-time audio output analysis
  - WebAudio only
- MIDI Enabled
  - Only with WebMIDI API enabled browsers (Google Chrome)
  - Hot plugging of MIDI devices is supported
- Collaborative app.
  - MIDI and shader inputs are not synchronized between users
- Feedback support
- Live coding/JIT compilation of shader code
- Global and per sessions settings automatic saving/loading
- No authentifications (sessions based)
- Per-sessions discussion system

1.3 System requirements

Fragment is a special kind of additive synthesizer which require a moderate deal of processing power in order to work properly, a medium-end GPU and CPU should work fine.

1.3.1 Browser

Fragment require a browser with full support for ECMAScript 5, CSS3, WebAudio and WebGL.

Well-tested and compatible browsers include Firefox 51.x and Chrome 55.x but Fragment may work with previous version of those browsers as well.

It is known to work on recent version of Safari and Opera as well.

Chrome or Chromium browser is recommended.

Fragment support the WebGL 2 API which improve performances and enable some advanced features which are only available if the browser support the WebGL 2 API.

1.3.2 CPU

Fragment may be quite hungry in term of computing resources, a dual core with high clock speed is recommended, a CPU with more cores can be useful if you just want to use the browser for audio output.

Several methods are provided to synthesize sounds, each with performances pros and cons:

- FAS (**recommended**):
  - Very fast
  - Multiple output support with soundcard choice
  - Many settings
  - Can run on a dedicated computer such as a Raspberry PI
  - Dedicated program which receive Fragment data over the network
- WebAudio oscillators:
  - Fastest under Chrome
  - May not work with Firefox
  - Require a fast CPU
  - Integrated global "release" envelope
- Wavetable (not recommended):
  - Most compatible browser method
  - Will produce crackles

1.3.3 GPU

Fragment was developed and tested with a NVIDIA GeForce GTX 970 GPU, a powerful GPU may be required if:

- you want higher score resolution
- you want to do visuals alongside audio
- you are doing complex things/use many inputs in your shader
- you want greater polyphonic/harmonics/partials capabilities

2 Concepts

Fragment is an additive synthesizer, it let you have full control over the timbral qualities of your sounds by the mean of GPU programming.

The biggest difference between Fragment and other synthesizers is that you work in the frequency domain instead of time domain, in that sense it is also a spectral synthesizer.

Unlike other additive synthesizer software, there is no need for knobs, sliders or any other controllers to sculpt your sounds, all of that is done by generating visuals which will determine your sounds harmonic content and dynamic characteristics.

There is many features in Fragment which allow any creative minds to produce complex sounds in different ways and even with external tools, i suggest all the readers to look at great softwares like the IanniX sequencer (`https://www.iannix.org`) and to use it with Fragment using the MIDI capabilities.

While Fragment may seem overwhelming at first, the only thing that is required to produce sounds with it is to know how to generate the visuals.

The goal of this section is to clarify the inner working of Fragment.

2.1 Additive synthesis

Fragment is first and foremost a powerful additive synthesizer which make an extended use of additive synthesis.

Additive synthesis is a sound synthesis technique that creates timbre by adding sine waves together.

Adding sine waves produce a timbre, the timbre quality is mainly defined by its harmonic content and the dynamic characteristics of the harmonic content.

The concept of additive synthesis is centuries old, it has first been used by pipe organs.

Fragment can theoretically produce any timbres with precise accuracy.

The only limit to the amount of sine waves that can be added by Fragment is the limit of the available processing power.

For example, on a Raspberry PI 3 (1.2GHz 64-bit quad-core ARMv8 CPU) 700 oscillators can be played simultaneously using two cores, matching the capability of the ANS synthesizer.

Fragment can also do other types of synthesis like substractive synthesis (additive synthesis in reverse).

2.2 The graphical score

Fragment graphical score represent the sound spectrum which is generated by the GPU from a fragment program.

The fragment program (also known as a fragment shader) is executed by the GPU and compute the color and other attributes of each "fragment" - a technical term which usually mean a single pixel.

The graphical score represent a kind of sonic canvas where the X axis represent time and the Y axis represent frequencies, you "paint" the graphical score by writing a fragment program which will be executed for each pixels of the graphical score.

What you hear in Fragment is determined by the position of "slices" which are added on the graphical score, slices are vertical bits of the graphical score which are merged together and produce an audible result.

The content of slices is captured at the rate of display refresh rate which conventionally should be 60fps most of the time.

The frequency mapping of the graphical score is fixed by a logarithmic formula, altough the formula cannot be changed right now, some parameters are available in the settings dialog to fine tune the frequency map.

The frequency mapping is defined by the formula:

$$f(y) = a * (2^{\frac{y}{n/o}})$$

Where:

- **a** is is the starting frequency
- **y** the vertical position
- **n** the number of oscillators (which is the height of the canvas)
- **o** the octave count

3 Sessions

Fragment sessions are isolated spaces which are open to all peoples who has access to the session name, they can be joined by going to the Fragment homepage (`https://www.fsynth.com`) or by typing the session name directly into the address bar as demonstrated below.

You can join any sessions directly from your browser address bar by replacing "yoursessionname" for the URL shown below by the session name you want to join or create:

https://www.fsynth.com/app/**yoursessionname**

Fragment store sessions content and settings on a server which mean that any of the synchronizable actions in the session such as the code editor content, canvas settings, slices and uniform inputs are automatically saved **if** Fragment is connected.

Fragment synchronize the code editor content, slices, canvas settings and uniform inputs across all the users in the session in real-time which mean that you can jam with your friends if you share the session URL.

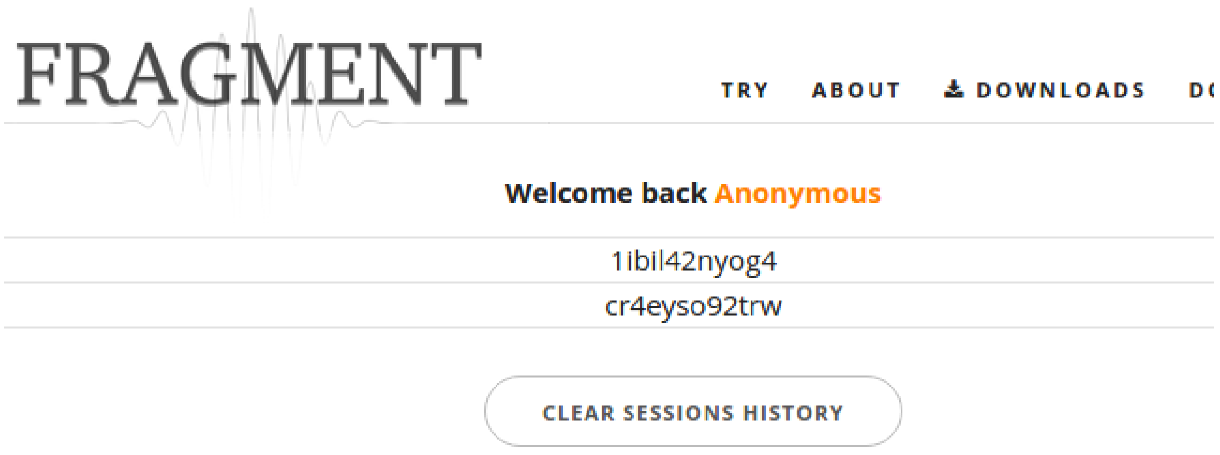
Note that MIDI note messages are not synchronized between users due to technical constraints but you can still add uniform inputs and assign MIDI devices to control them, their values are synchronized between users.

Some settings are saved locally on your browser, some of them are global (like the username and settings related to the editor) and some of them are saved per sessions such as the MIDI settings and gain.

Fragment is respectful of the user and does not store any informations related to the user outside the boundary defined by the application focus.

3.1 Homepage and sessions history

The Fragment homepage (`https://www.fsynth.com`) can be used to retrieve the various sessions that you joined, a session list will be shown once you joined at least one session.



Welcome back Anonymous

1ibil42nyog4

cr4eyso92trw

(images/homepage_sessions.png) You can join a session from the list by clicking on it and by clicking on the door icon.

You can remove a session from the list by clicking on it and by clicking on the trash icon.

You can also clear the sessions history by clicking on the corresponding button, it will not delete the session content, just your settings for that session and the session will removed from the list, a deleted session can be joined back from the homepage form or by the browser address bar if you know its name.

4 FAS

FAS (Fragment Audio Server) is the Fragment synthesis engine as a server, it is an external independent program written with the C language for very fast real-time audio synthesis.

It is recommended to use Fragment with FAS enabled or use the standalone version of Fragment which include FAS.

The advantage of using FAS is that it provide the best audio performances along with audio device choice, multiple audio output per slices, sample rate choice and realtime scheduling under Linux.

Fragment communicate with FAS by sending data through the network when the FAS option is enabled, FAS can run on any machines independently of the Fragment application, it gather the pixels data from Fragment in real-time, compute things and deliver the audio through the selected audio device.

FAS can be downloaded on the Fragment homepage (`https://www.fsynth.com`), it can also be compiled from sources on GitHub (`https://github.com/grz0zrg/fas`).

FAS is provided as an executable for the following platforms at the moment:

- Windows x86/x64 32/64 bits
- Linux x86/x64 32/64 bits
- Raspberry PI (700 oscillators can play at the same time under optimized system)

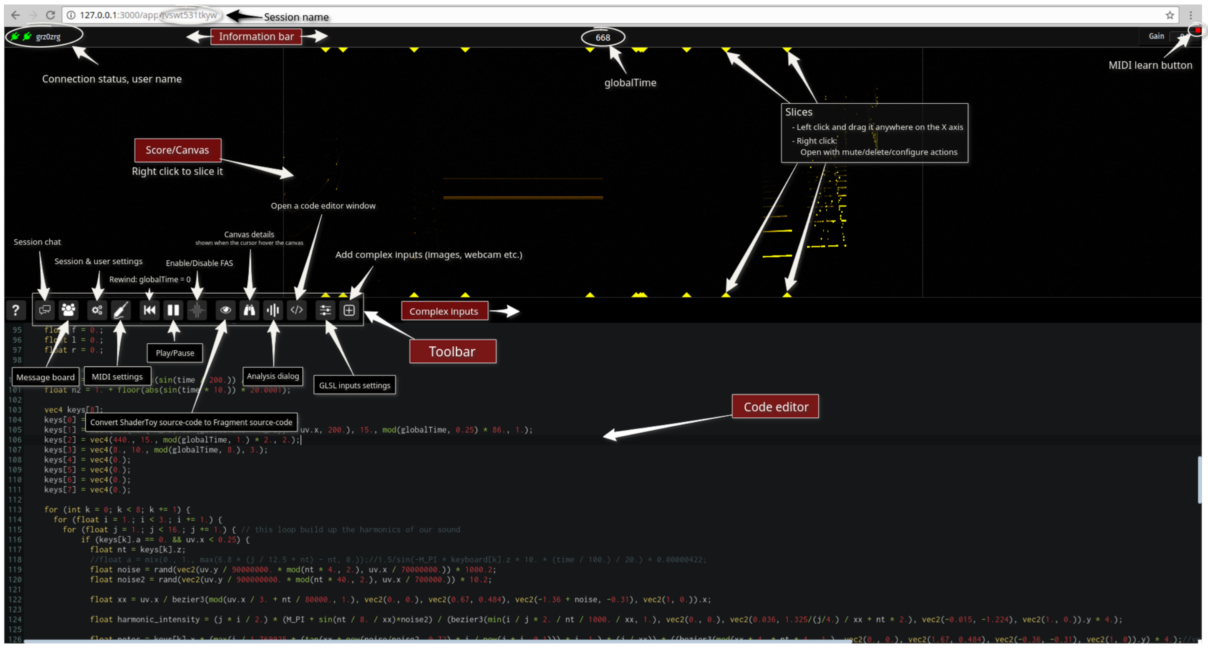
Here is a list of the program arguments which FAS support (along with the default settings if not provided):

- –h
    - show the program arguments

- –i
  - print audio device informations
- –device -1
  - the id of the audio device to use (running FAS without arguments once will print the devices found along with their id and informations)
- –iface 127.0.0.1
  - the address on which to listen
- –port 3003
  - the network port on which to listen
- –output_channels 2
  - the number of output channels, the value must be an even number
- –alsa_realtime_scheduling 0
  - enable/disabled ALSA realtime scheduling, a Linux only parameter
- –sample_rate 44100
  - can be any sample rate supported by the selected audio device
- –frames 512
  - the audio buffer size, this have an effect on audio latency and CPU usage
- –wavetable_size 8192
  - the wavetable size, the default value should be enough for most uses
- –fps 60
  - this setting is provided in case Fragment does not capture slices at 60 fps, on most setup Fragment try to capture slices at 60 fps
- –ssl 0
  - this is provided to use FAS over SSL (not recommended at all!)
- –max_height 4096
  - if you set the score height above 4096 pixels in Fragment, keep in mind that you must increase this!
- –deflate 0
  - a value of 1 will enable deflate compressed packets (not recommended)
- –rx_buffer_size 4096
  - FAS will fragment packets which are bigger than the default value
- –frames_queue_size 7
  - this is the length of the maximum number of frames waiting in the queue
  - this is an important parameter, if you increase this too much the audio will be delayed
- –commands_queue_size 16
  - this is the same as frames_queue_size except for commands (gain changes etc.)
  - should be a power of 2 and positive integer

Once run, you can stop the FAS application by pressing any keys.

5 The user interface



```
    float f = 0.;
    float l = 0.;
    float r = 0.;

    float n2 = 1. + floor(abs(sin(time * 10.)) * 20.0001);

    vec4 keys[8];
    keys[0] =
    keys[1] =                                         uv.x, 200.), 15., mod(globalTime, 0.25) * 86., 1.);
    keys[2] = vec4(440., 15., mod(globalTime, 1.) * 2., 2.);
    keys[3] = vec4(8., 10., mod(globalTime, 8.), 3.);
    keys[4] = vec4(0.);
    keys[5] = vec4(0.);
    keys[6] = vec4(0.);
    keys[7] = vec4(0.);

    for (int k = 0; k < 8; k += 1) {
      for (float i = 1.; i < 3.; i += 1.) {
        for (float j = 1.; j < 16.; j += 1.) { // this loop build up the harmonics of our sound
          if (keys[k].a == 0. && uv.x < 0.25) {
            float nt = keys[k].z;
            //float a = mix(0., 1., max(6.8 * (j / 12.5 + nt) - nt, 0.));//1.5/sin(-M_PI * keyboard[k].z * 10. * (time / 100.) / 20.) * 0.00000422;
            float noise = rand(vec2(uv.y / 90000000. * mod(nt * 4., 2.), uv.x / 70000000.)) * 1000.2;
            float noise2 = rand(vec2(uv.y / 900000000. * mod(nt * 40., 2.), uv.x / 700000.)) * 10.2;

            float xx = uv.x / bezier3(mod(uv.x / 3. + nt / 80000., 1.), vec2(0., 0.), vec2(0.67, 0.484), vec2(-1.36 + noise, -0.31), vec2(1, 0.)).x;

            float harmonic_intensity = (j * i / 2.) * (M_PI + sin(nt / 8. / xx)*noise2) / (bezier3(min(i / j * 2. / nt / 1000. / xx, 1.), vec2(0., 0.), vec2(0.036, 1.325/(j/4.) / xx + nt * 2.), vec2(-0.015, -1.224), vec2(1., 0.)).y * 4.);
```

(images/ui_help.png)

Click to view the full image

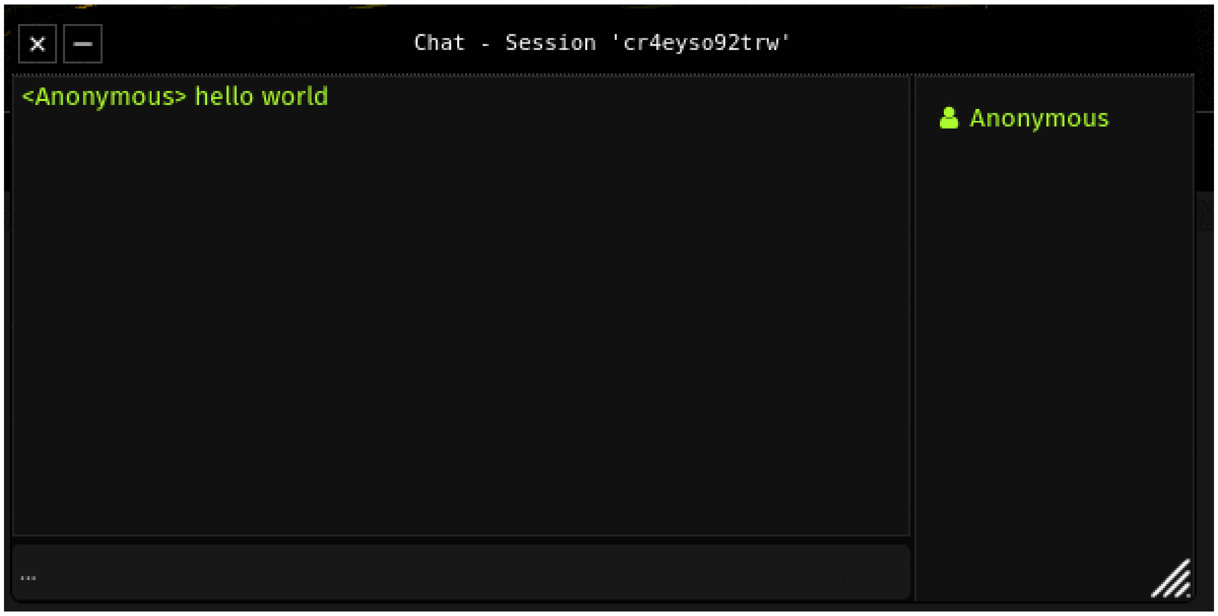
(images/ui_help.png) Fragment user interface is quite easy to get once you figure out its layout.

It is composed of a few parts with very specific roles, here is the list of parts along with detailed informations of their functions and supported actions.

5.1 Dialogs

There is many dialogs in Fragment, they are composed of a title bar with some actions, the dialog content and sometimes a status bar, dialogs hover above the rest of the application, you can move them around and close them, some are also resizable, minimizable and detachable in a separate window.

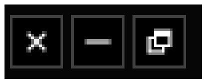
To resize them, place your mouse cursor on the bottom left corner of the dialog, click and drag to the desired size.



<Anonymous> hello world

A minimized dialog will take less space on the screen by only displaying the title bar.

Here is a list of dialog actions (in the order by which they appear from left to right):



(images/dialog_actions.png)

- close the dialog
- minimize/maximize
- detach the dialog in a new window

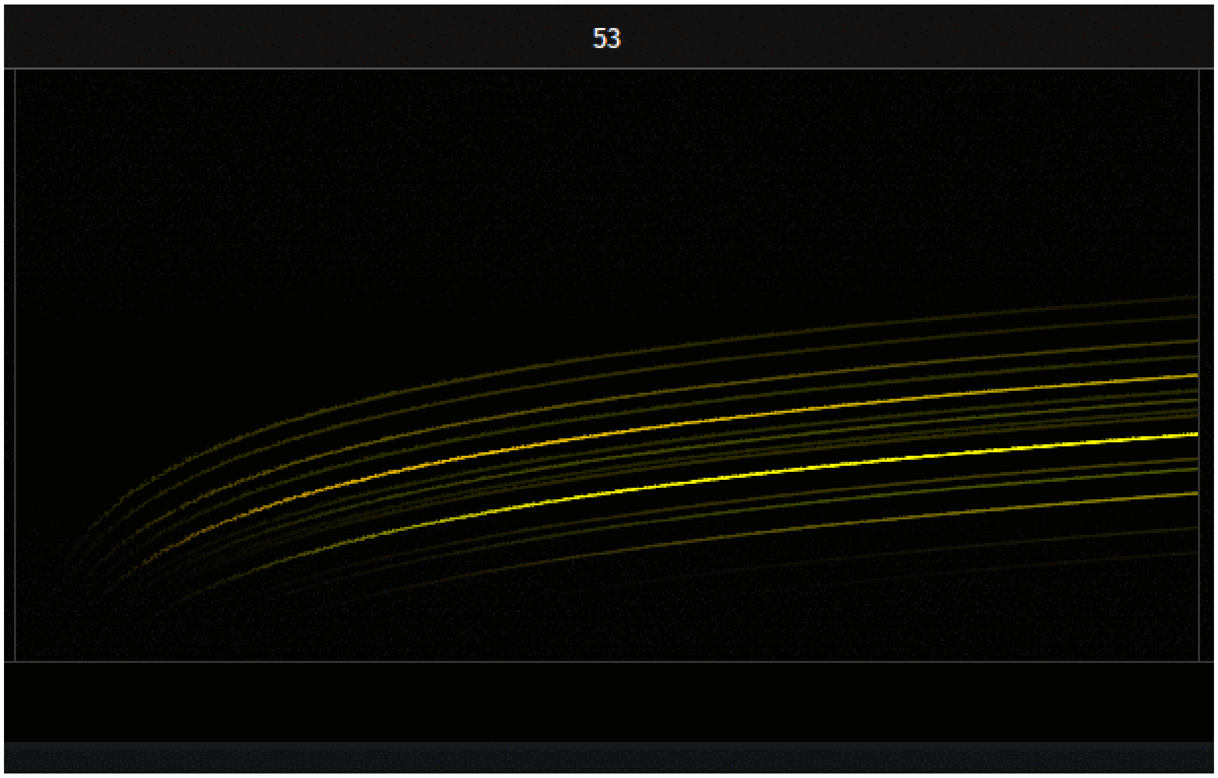
5.2 infobar

(The information bar) The information bar at the top convey minor and major informations such as (in order from left to right):

- connectivity status
  - a red indicator signal that the connection to this service failed
  - the service name can be found by hovering the cursor on the icon
- your online name
  - you can change your username by clicking on it
- various informations which can be enabled or disabled in the settings dialog such as:
  - the current frequency under the mouse cursor
  - the actual number of simultaneous MIDI notes
  - the actual number of oscillators playing
- the playback time
- a gain controller (master volume)

5.3 The graphical score

You can slice the graphical score by right-clicking on it and clicking on the + icon, this will add a vertical bar which will capture that part of the canvas.



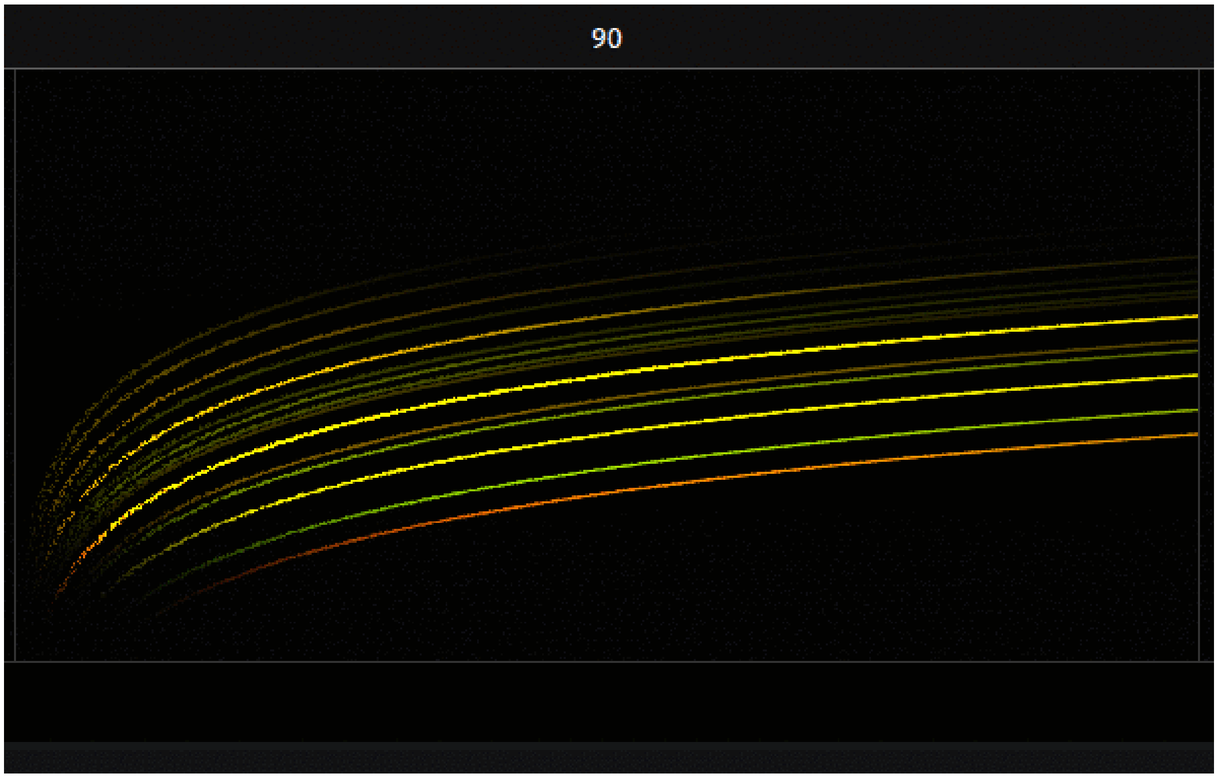
You are free to add any number of slices, adding many slices may have an impact on performances.

5.3.1 Slice

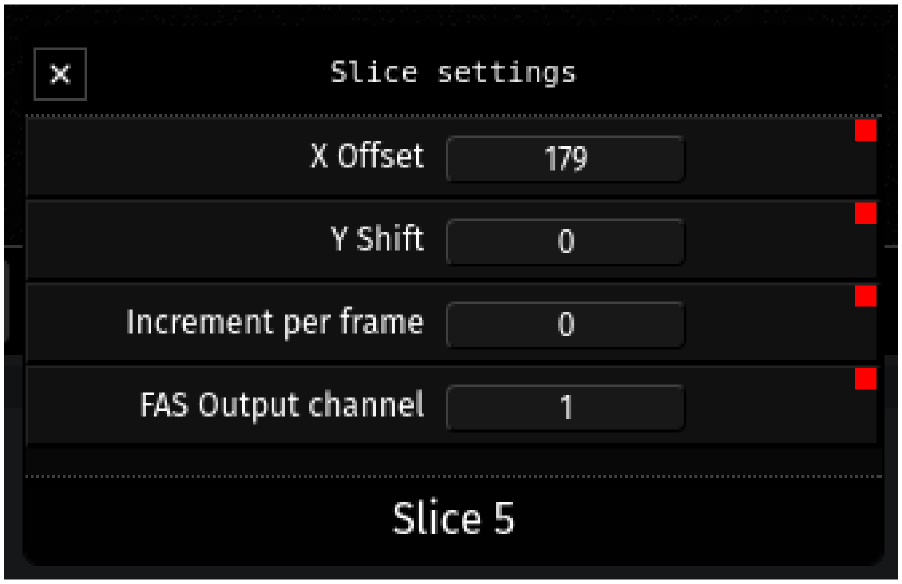
Slices are an important part of Fragment, there will be no sounds if you don't add at least one slice.

The purpose of slices is to capture the pixels of vertical parts of the canvas which will be fed to the additive synthesis engine, they are like turntable needles, they can be dragged around in real-time by the mouse or by a MIDI controller, you can use programs like IanniX (`https://www.iannix.org`) to move them following certain kind of patterns.

Slices can be moved by dragging them on the X axis, to do so, maintain the left mouse button on a slice and move the mouse cursor around on the horizontal axis.



Double-clicking on a slice open its settings dialog directly:


Slice settings

X Offset	179
Y Shift	0
Increment per frame	0
FAS Output channel	1

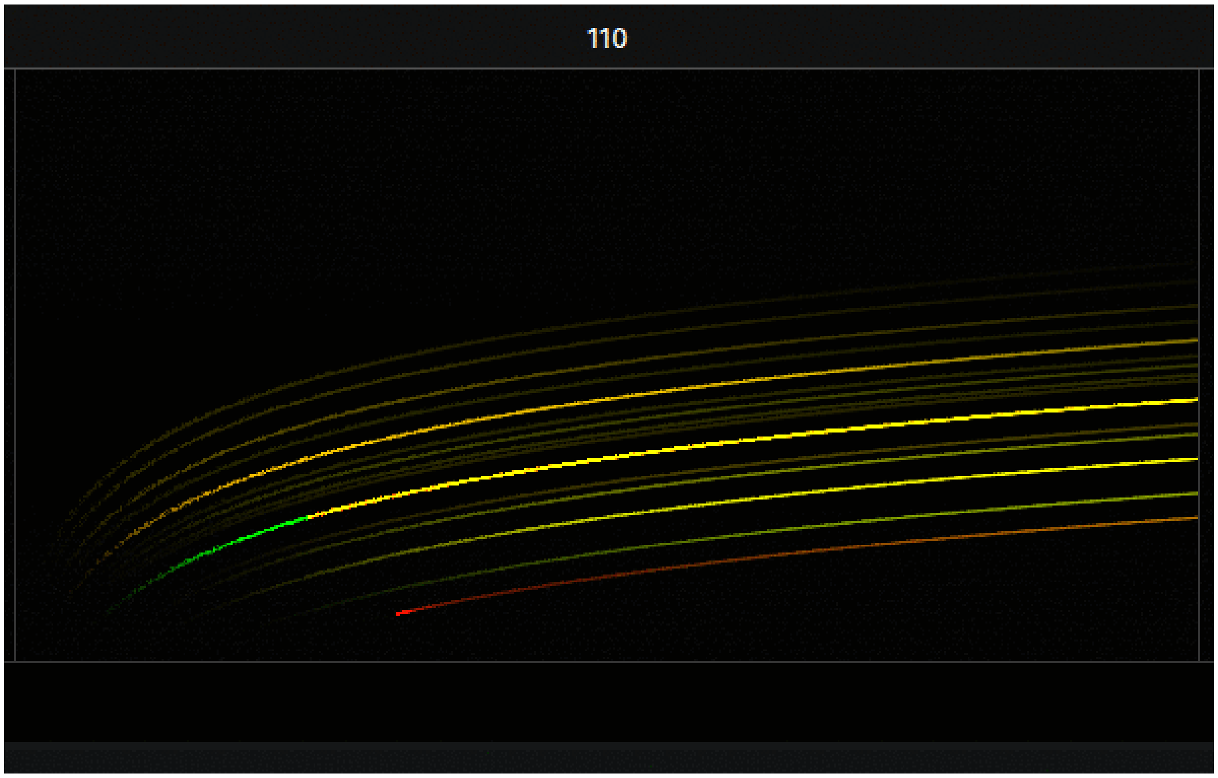
Slice 5

(images/slice_settings.png) The following actions are possible by right-clicking on a slice:

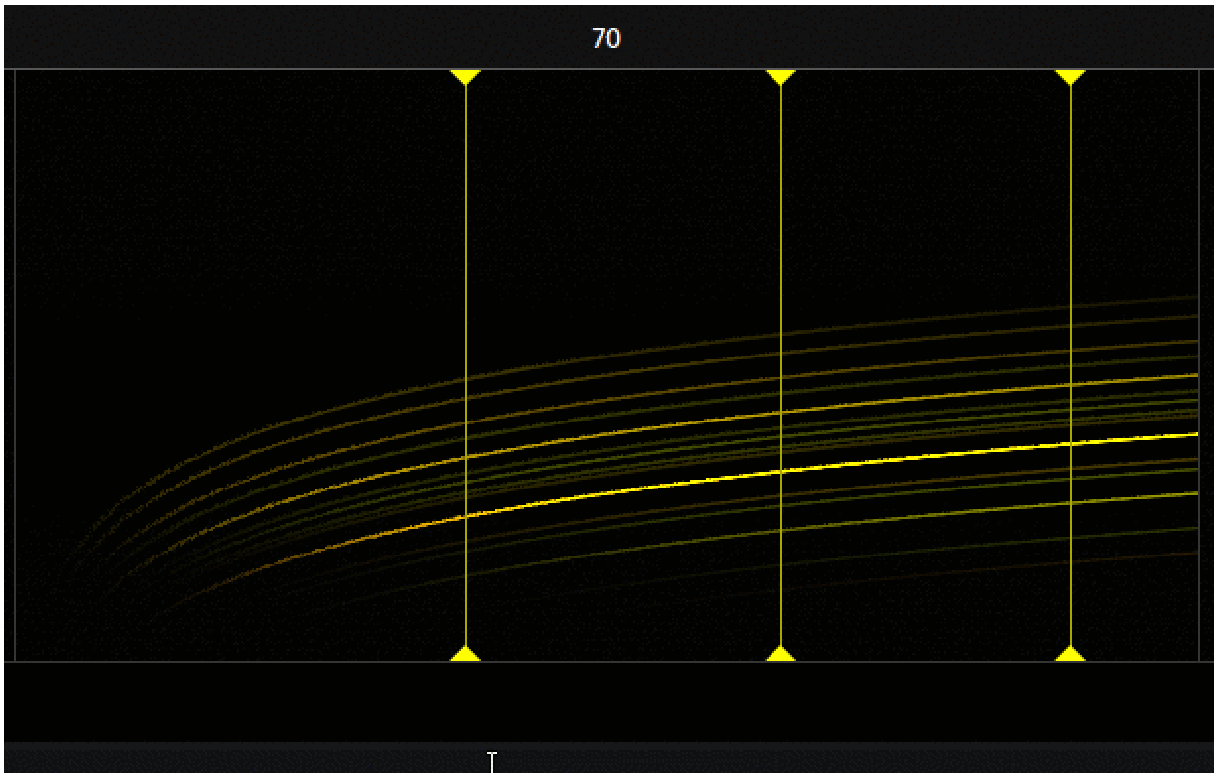
- mute/unmute
  - the synthesis engine will ignore a muted slice
- open the slice settings which posses the following controllers:
  - X Offset: the slice horizontal position which can be controlled by a MIDI controller

    - Y Shift: pitch the slice audio up or down (there is no visual representation of this)
    - Increment per frame: This allow the slice to move left or right automatically, this is the increment value per frames
    - FAS Output channel: the l/r output channel which will be used by FAS for that slice
- deletion

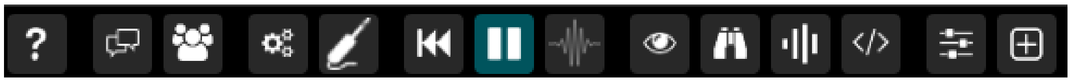
Here is how you can mute a slice:



Here is how you can delete a slice:



5.4 The toolbar



(`images/toolbar.png`)
The toolbar is a collection of tools and settings which are grouped by sections, here is a detailed list of the toolbar items (in order from left to right):

- help
- social
  - session live chat
  - direct link to the community board
- settings
  - session and global settings
  - MIDI settings
- transport
  - reset playback time to 0
  - play/pause
  - FAS enable/disable
- helpers
  - ShaderToy (`https://www.shadertoy.com`) converter
  - canvas axis details which appear when the canvas is hovered by the cursor
  - analysis dialog (not working when FAS is enabled)
  - clone the code editor in a separate window

- fragment inputs
    - uniforms
    - add complex inputs

5.5 The fragment inputs

Fragment inputs are complex inputs which can be used in the fragment program as a 2D texture.

The fragment inputs panel is a list of the added complex inputs, each of them appear as a thumbnail near the **add complex inputs button**, nothing will be shown if no complex inputs were added.

All complex inputs can be used as 2D textures (texture2d) in the fragment program, they are defined as **iInputN** where N is the id of the input starting from 0.

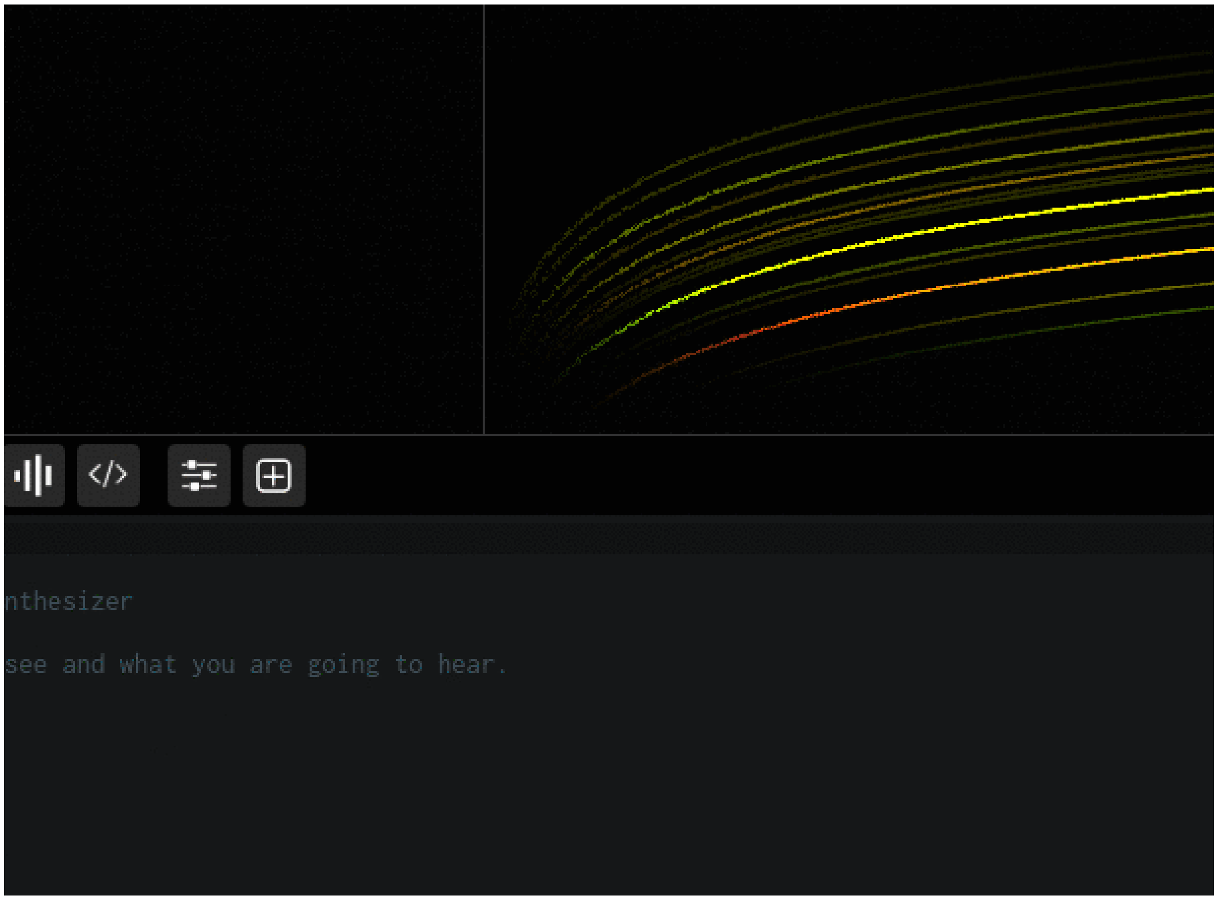
You can find the input id by hovering over the thumbnail or in the title of the complex input settings dialog.

You can add one by clicking on the "add complex inputs" button, here is a list of the available complex inputs:

- image file
- webcam
    - allow the real-time webcam video to be used in the fragment program
- audio file

By right clicking on a the complex input thumbnail, the following actions appear:

- delete
- complex input settings dialog


nthesizer

see and what you are going to hear.

The complex input settings dialog have several options:

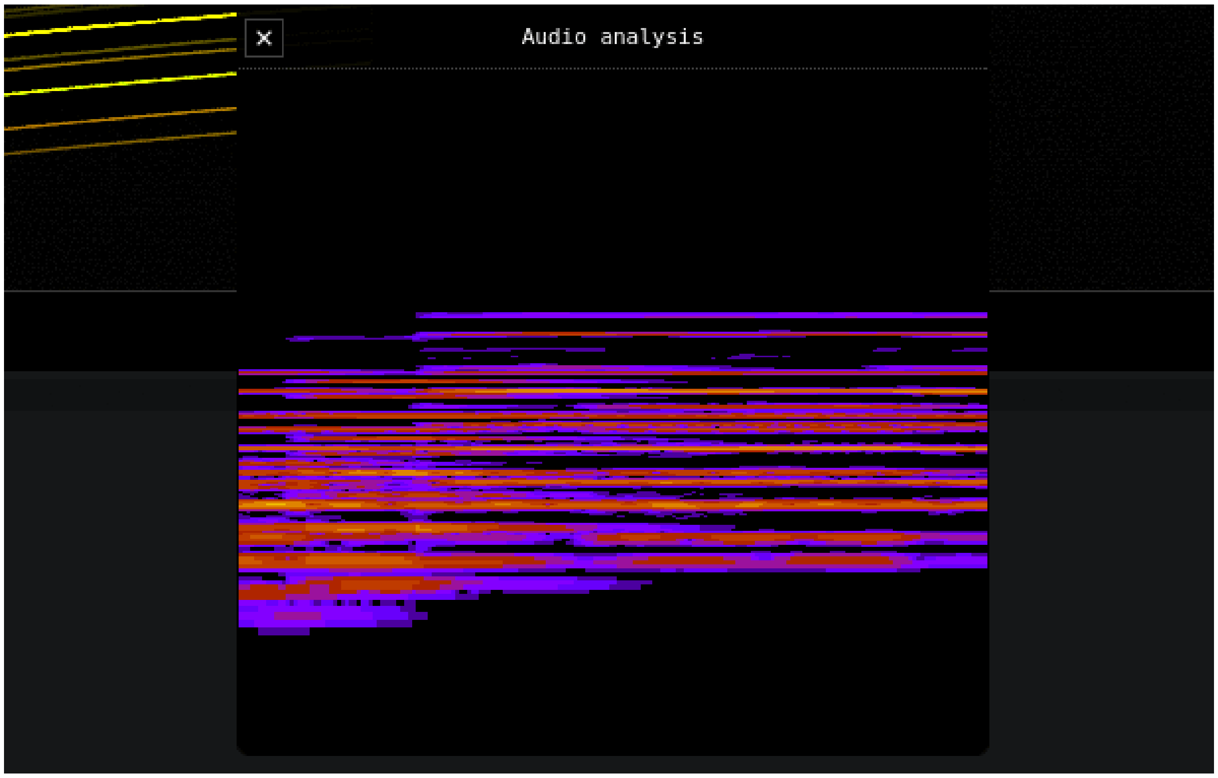
- filter
    - nearest: no interpolation
    - linear: linear interpolation
- Wrap S
    - clamp: stop horizontally when outside the (0,0) to (1,1) range
    - repeat: repeat horizontally when outside the (0,0) to (1,1) range
    - mirrored repeat: same as repeat but mirrored
- Wrap T
    - clamp: stop vertically when outside the (0,0) to (1,1) range
    - repeat: repeat vertically when outside the (0,0) to (1,1) range
    - mirrored repeat: same as repeat but mirrored
- VFlip: flip the texture vertically

Note: The "repeat" and "mirrored" Wrap S/T option will be unavailable if your browser does not support the WebGL 2 API, it is only available by adding images with power-of-2 dimensions (128x64, 256x256 etc.) or by using a browser with WebGL 2 support.

5.6 Analysis dialog

The analysis dialog is a real-time spectogram view of the audio output.

Note: When FAS is enabled, no spectrogram will be shown.

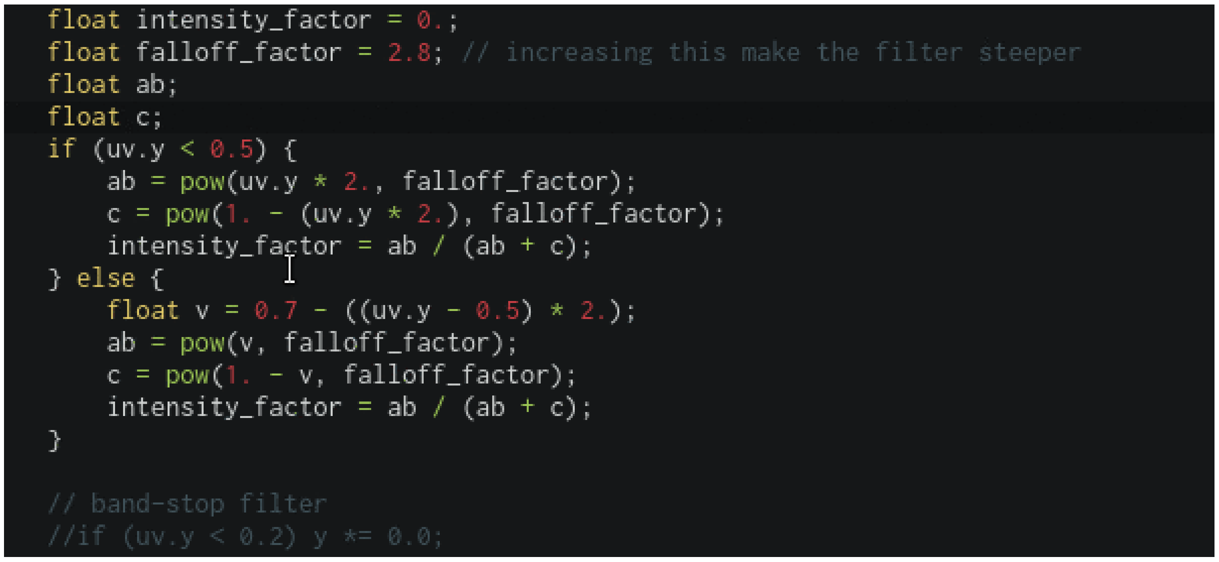


5.7 The code editor

The fragment editor is one of the most important tool of Fragment since it allow the user to generate the visuals which are fed to the additive synthesis engine.

GLSL (`https://www.khronos.org/files/opengles_shading_language.pdf`) code is what you type in the code editor to generate the visuals.

The fragment program is compiled as you type, if the compilation fail, the code editor will notice you with a floating message and with a red message at the lines that cause the compilation to fail, all of that without interrupting sounds/visuals output, this enable powerful live coding.



```
float intensity_factor = 0.;
float falloff_factor = 2.8; // increasing this make the filter steeper
float ab;
float c;
if (uv.y < 0.5) {
    ab = pow(uv.y * 2., falloff_factor);
    c = pow(1. - (uv.y * 2.), falloff_factor);
    intensity_factor = ab / (ab + c);
} else {
    float v = 0.7 - ((uv.y - 0.5) * 2.);
    ab = pow(v, falloff_factor);
    c = pow(1. - v, falloff_factor);
    intensity_factor = ab / (ab + c);
}

// band-stop filter
//if (uv.y < 0.2) y *= 0.0;
```

The changes that you make in the code editor are automatically saved per named sessions, changes are also synchronized in real-time between all the users of the session you are currently in, this is the collaborative nature of Fragment.

The code editor is powered by the CodeMirror library, it feature many neat things such as:

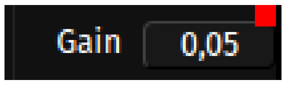
- GLSL syntax highlighting
- highlight matches
- brackets/parentheses opening/end highlights
- fullscreen editor by pressing F11
- integrated compilation errors/messages (does not interrupt sounds/visuals)
- line numbers
- many bundled editor themes

Some of the code editor features can be enabled/disabled in the global settings dialog.

If you experience audio stuttering as you type, it is recommended to detach the code editor off the main window, due to the complexity of the web and the complexity of the Fragment application, many things are not as optimal as they should be, you may hear audio crackles due to browser reflow.

5.8 MIDI learn functionality

MIDI learn is a neat feature of Fragment which enable any MIDI learn enabled widget to be controlled by a MIDI controller.



(images/midi_learn.png)

The red square indicate MIDI learn functionality support for this widget

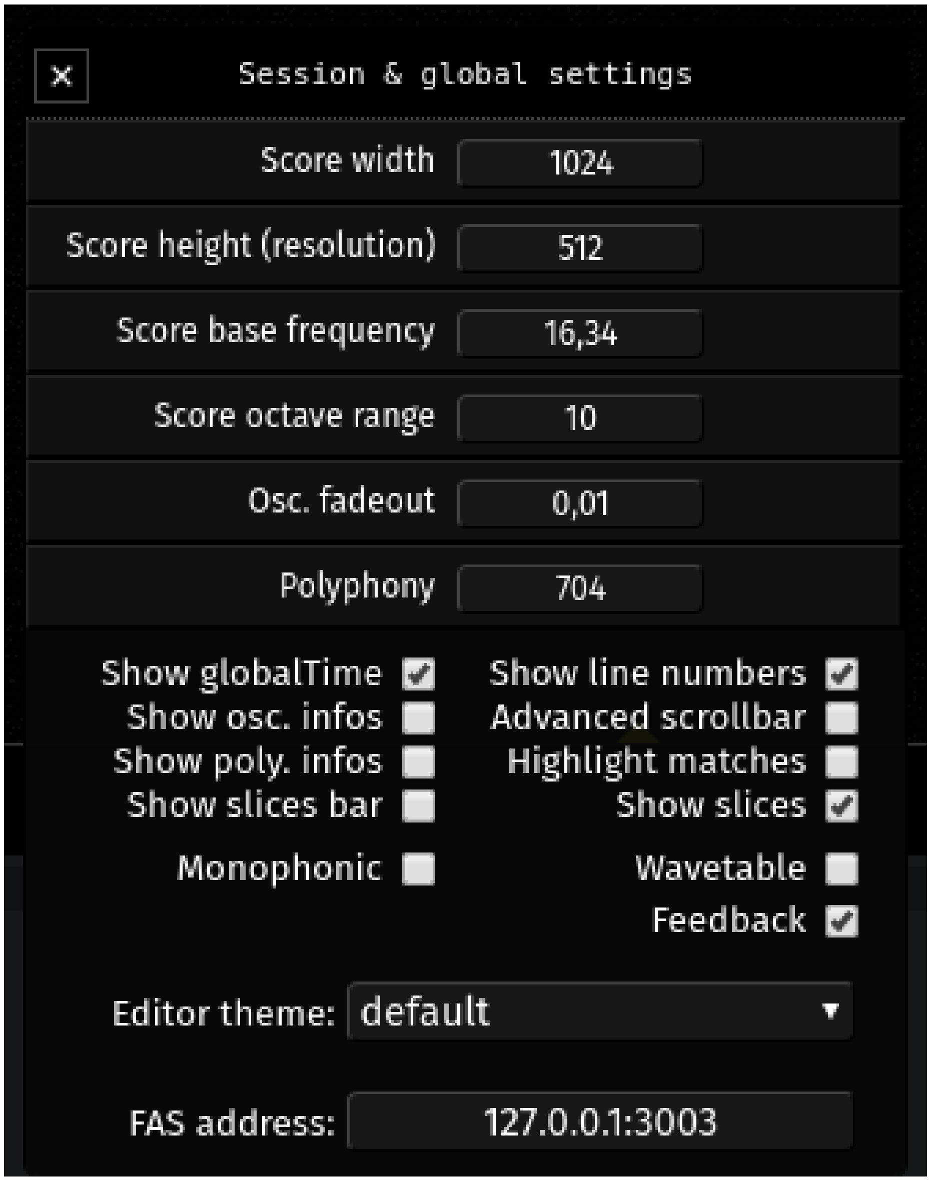
The red square appearing on an UI interface widget indicate that the MIDI learn functionality is supported for the widget, it only appear on WebMIDI enabled browsers such as Chrome and Opera and on widgets which are allowed to be controlled by MIDI.

By left clicking on the red square, it turn green and any inputs from the enabled MIDI devices will be captured by the widget.

Once the MIDI input is captured, the green square become red again (which is a locked state) and the MIDI control will be assigned to the widget.

It is possible to reset the MIDI control assigned to the widget by clicking on the red square and clicking again on the green square aka double clicking.

5.9 The session/global settings dialog



Score width 1024
Score height (resolution) 512
Score base frequency 16,34
Score octave range 10
Osc. fadeout 0,01
Polyphony 704

Show globalTime ✓
Show osc. infos ☐
Show poly. infos ☐
Show slices bar ☐
Monophonic ☐
Show line numbers ✓
Advanced scrollbar ☐
Highlight matches ☐
Show slices ✓
Wavetable ☐
Feedback ✓

Editor theme: default ▾

FAS address: 127.0.0.1:3003

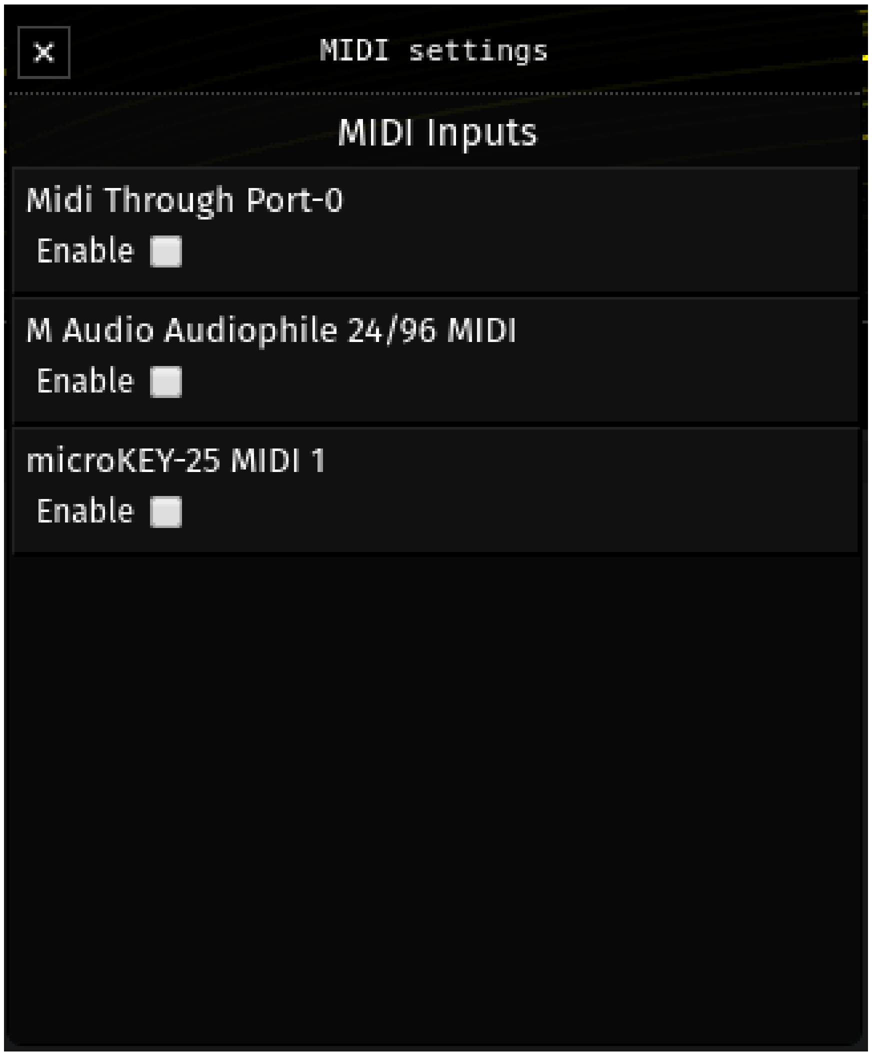
(images/settings.png)

The session & global settings dialog content (in order from top to bottom):

- Score width

  - The score width in pixels units
- Score height
  - The score height in pixels units
  - Higher height = better resolution in term of frequencies
- Score base frequency
  - Determine the base frequency in hertz units
- Score octave range
  - Control the range of frequencies
- Osc. fadeout
  - WebAudio fadeout time (like "release" parameter in other synts, WebAudio oscillators only)
- Polyphony
  - Maximum polyphony
- Show globalTime
  - Hide/Show the globalTime in the informations bar
- Show osc. infos
  - Hide/Show the number of oscillators playing simultaneously in the informations bar
- Show poly. infos
  - Hide/Show the polyphony infos for per output channels in the informations bar
- Show slices bar
  - Hide/Show slices vertical bar
- Show line numbers
  - Hide/Show the line number in the code editor
- Advanced scrollbar
  - Enable/disable a better looking scrollbar for the code editor
- Highlight matches
  - Enable/disable matches highlight in the code editor (when something is selected)
- Show slices
  - Hide/Show slices, can be useful for visuals, this settings is not saved
- Monophonic
  - Enable/disable monophonic mode
  - If monophonic is enabled, only the alpha value is used by the synthesis engine, the full RGB output can then be used for visuals
- Wavetable
  - Enable/disable the synthesis engine wavetable feature (uglier audio output but may be the less CPU intensive mode)
- Feedback
  - Enable/disable the availability of the previous frame in the fragment shader (aka feedback), this is useful for complex effects, may be CPU/GPU intensive
- Editor theme
  - A list of themes for the code editor
- FAS address
  - The location of FAS (Fragment Audio Server) on the network

5.10 MIDI settings dialog



Midi Through Port-0
Enable ☐

M Audio Audiophile 24/96 MIDI
Enable ☐

microKEY-25 MIDI 1
Enable ☐

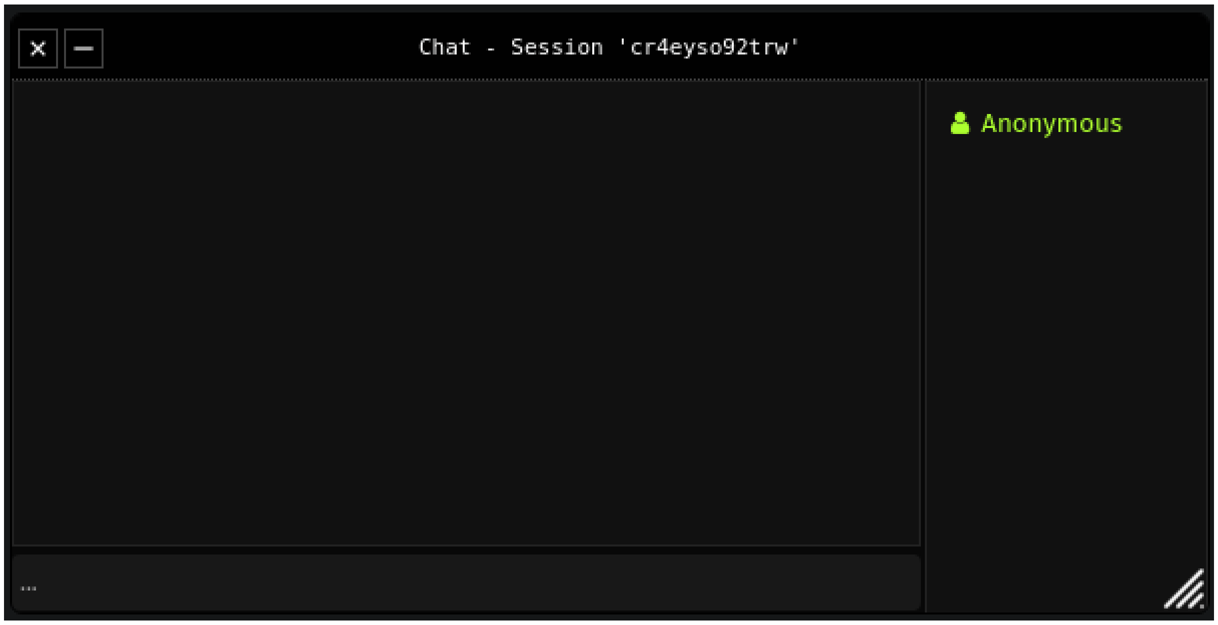
(`images/midi_settings.png`)
The MIDI settings dialog show all the MIDI input devices found at this time, by default there is no MIDI input devices enabled, you can enable MIDI devices by checking the checkbox below the MIDI device name.

Once enabled, Fragment will receive any MIDI messages from that device.

Fragment support hot plugging of MIDI devices, any MIDI devices which are plugged or unplugged while Fragment is running will be added or removed in the MIDI settings dialog.

Fragment keep track of your MIDI settings choice for particual devices per sessions, this mean that if a MIDI device is enabled, when you quit Fragment or that the MIDI device is unplugged and you launch again Fragment and that the MIDI device is plugged in, Fragment will enable it automatically.

5.11 Session chat dialog



(`images/session_chat.png`) The session chat dialog allow discussions with all users in the current session.

It is a simple but effective chatbox that you can resize, move around or detach. it has three parts:

- messages list
- users list
- input

The green user in the user list indicate yourself.

You can send a message to all users in the session by clicking on the input then typing your message and pressing ENTER.

5.12 Uniforms dialog

The uniforms dialog is a very powerful functionality.

It enable you to define fragment program uniforms (aka variables) which are synced with all users in the session and are controllable by any MIDI input devices enabled.

All the uniforms added will be shown in the uniforms dialog and will be defined automatically in the fragment program, you can then just use the symbol name in the code editor.

You can add the following scalar uniforms types:

- bool
- int
- float

If count is higher than 1, Fragment will add the scalar uniforms as an array.

You can also add vector uniforms of the following types (note: the components field determine the number of components for the following uniform type):

- bvec2

- bvec3
- bvec4
- ivec2
- ivec3
- ivec4
- vec2
- vec3
- vec4

If count is higher than 1, Fragment will add the vector uniforms as an array.

6 How to produce sounds by programming your GPU

Fragment usage of the GPU is restricted to a single fragment program which will be executed by the GPU for each pixels of the graphical score.

The fragment program is written in GLSL (OpenGL Shading Language) which has a syntax quite similar to the C language but is much simpler to learn.

You can do anything from ray-traced 3D to drawing simple lines and define their behaviors with Fragment, the only requirement is high school mathematics.

This documentation focus on the audio synthesis aspect of Fragment as there is already plenty resources covering the creation of stunning visuals with GLSL programs on the web.

There is also many applications that let you create stunning visuals in your browser by the same method, one of the most popular one and compatible with Fragment (by using the convert ShaderToy button of the toolbar) is ShaderToy (`https://www.shadertoy.com/`), this program let you build visuals and audio at the same time, just like Fragment but with a more conventional approach to audio synthesis.

6.1 OpenGL Shading Language

The Khronos Group (authors of the language specification) released several reference cards of the GLSL specification, the reference cards are compact reference of the full language specification which can be used in conjunctions with code samples to learn the OpenGL Shading Language quickly.

Since there is plenty of resources to learn the OpenGL Shading Language, the documentation provided in this section will only provide the official reference cards which are enough to learn from to understand all the GLSL code that will follow.

As Fragment can use WebGL 2.0 if your browser has support for it, the reference cards for both WebGL 1.0 GLSL and WebGL 2.0 GLSL (more functionalities) is provided, keep in mind that if Fragment run fine with your browser, you can safely use the WebGL 1.0 GLSL reference card as a starting point:

- WebGL 1.0 OpenGL Shading Language
  - Page 1 PDF (`pdf/webgl1_glsl_1.pdf`)
  - Page 2 PDF (`pdf/webgl1_glsl_2.pdf`)
- WebGL 2.0 OpenGL Shading Language
  - Page 1 PDF (`pdf/webgl2_glsl_1.pdf`)
  - Page 2 PDF (`pdf/webgl2_glsl_2.pdf`)
  - Page 3 PDF (`pdf/webgl2_glsl_3.pdf`)

Here is a simple example of GLSL code that Fragment accept which just set all pixels to black:

```
void main () {
  gl_FragColor = vec4(0., 0., 0., 0.);
}
```

In the subsequent GLSL code, blue words are GLSL keywords, red are float values and green words are pre-defined specific definitions or functions.

6.2 Pre-defined uniforms

Fragment has many pre-defined uniform variables which can be used to animate/modulate things, here are all the pre-defined uniforms along with their type that can be used directly in your code:

- **vec2** resolution
  - score/viewport resolution in pixels units
- **float** globalTime
  - playback time in seconds
- **float** baseFrequency
  - score base frequency
- **float** octave
  - score octave range
- **vec4** mouse
  - the first two components is the normalized mouse pixel coords updated when the left mouse button and the cursor is moving over the canvas
  - the second two components is the normalized mouse pixel coords only updated when the left mouse button is down on the canvas
- **vec4** date
  - year
  - month
  - day
  - time in seconds
- **sampler2D** iInput**N**
  - typical usage: texture2D(iInput0, uv)
- **sampler2D** pFrame
  - this uniform will be available when feedback is enabled and contain the previous frame
  - typical usage: texture2D(pFrame, uv)
- **vec4N** keyboard
  - note-on frequency
  - note-on velocity
  - note-on elapsed time
  - note-on MIDI channel
- **function** htoy
  - take a frequency as argument and return its vertical position on the canvas (in pixels units)
- **function** fline
  - take a frequency as argument and return either 1 or 0, this is a shortcut to draw a line at a specified frequency, it use the htoy function internally

6.3 The timbre

It is difficult to define precisely the attributes of a complex sound, simplistic models would say that all complex sounds are a combination of a fundamental frequency and the harmonics which are integer multiple of the fundamental frequency but as you can find on the Wikipedia article on timbre (`https://en.wikipedia.org/wiki/Timbre`), there is many more attributes.

Here is an excerpt of timbre major attributes by Robert Erickson taken from the Wikipedia page:

- The range between tonal and noiselike character
- The spectral envelope
- The time envelope in terms of rise, duration, and decay
- The changes both of spectral envelope (formant-glide) and fundamental frequency (micro-intonation)
- The prefix, or onset of a sound, quite dissimilar to the ensuing lasting vibration

6.4 Drawing the fundamental

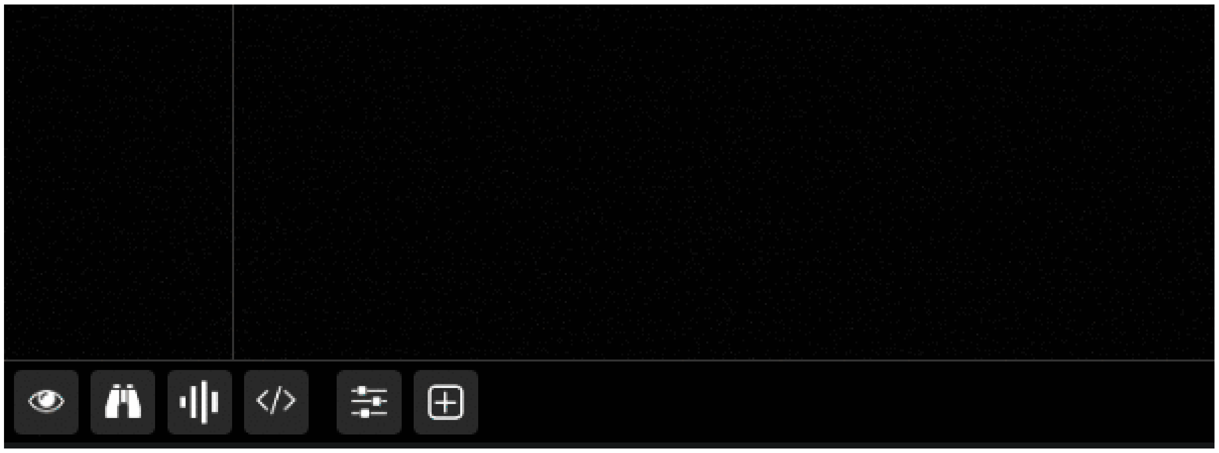
The fundamental is the simplest constituent of our timbre (its pitch and lowest frequency) and also the fundamental basis for building the harmonics (also called overtones) etc.

We will produce a fundamental by drawing a horizontal lines and later on its harmonics, all of that will constitute all the frequencies contained by our sound (aka harmonics, partials etc.), those horizontal lines will have several conditional transformations and other processing applied to them later on to produce complex sounds.

Here is how to draw a horizontal line which will produce a fundamental 440Hz sine wave:

```
void main () {
  float frequency = 440.;
  float l = 0.;
  float r = 0.;

  // because this code is executed for each fragments
  // the horizontal line is defined mathematically
  // by a simple pre-defined function called fline:
  // step(abs(gl_FragCoord.y - htoy(frequency)), 0.5);
  // which return 1 or 0, the color
  // for the left and right channel aka red and green output color
  l += fline(frequency);
  r += fline(frequency);

  // plot the line
  // Fragment use the RED and GREEN component
  // for the LEFT and RIGHT audio channel in stereophonic mode
  // if monophonic mode is enabled, the alpha component will be used for the audio
  // in this case all other components are unused
  // thus the blue component can be used for visuals or debugging etc.
  gl_FragColor = vec4(l, r, 0., 0.);
}
```

You can verify that the produced horizontal line is a 440Hz sine wave by enabling the axis details tool in the toolbar and point your cursor on the horizontal line:



You might be surprised to see that this is actually not exactly a 440Hz sine wave, the value is rounded due to the way frequencies are mapped, this can be fixed by increasing the score height to get a more precise mapping, this will require higher processing power however.

6.5 Drawing harmonics

Harmonics (also called overtones) are defined as positive integer multiple of the frequency of the fundamental.

We will extend the previous code by adding a loop:

```
void main () {
  float l = 0.;
  float r = 0.;
  float base_frequency = 440.;

  // the number of harmonics that we want (including the fundamental)
  // we define it as "const" because the current GLSL spec.
  // does not allow dynamic variables for loops index
  const float harmonics = 8.;

  // draw a horizontal line for each harmonics
  // including the fundamental
  for (float i = 1.; i < harmonics; i += 1.) {
    // multiply the fundamental frequency by the index of the current harmonic
    l += fline(base_frequency * i);
    r += fline(base_frequency * i);
  }

  gl_FragColor = vec4(l, r, 0., 0.);
}
```

The produced timbre is actually too raw because all our harmonics have the same value...

We can fix that by attenuating the higher frequencies, this is similar to the concept of filters in common synthesizers, this is really important as harmonics amplitudes is an important parameter for timbre quality, for example, by decreasing the amplitude of our harmonics gradually, we can create a sawtooth wave, an exponential attenuation provide good result generally in audio due to the way humans brain interpret sounds, we could also just attribute different intensity for each of our harmonics:

```
void main () {
  float l = 0.;
  float r = 0.;
  float base_frequency = 440.;

  float attenuation_constant = 1.95;

  const float harmonics = 8.;

  for (float i = 1.; i < harmonics; i += 1.) {
    // exponentially attenuate higher frequencies
    float a = 1. / pow(i, attenuation_constant);

    l += fline(base_frequency * i) * a;
    r += fline(base_frequency * i) * a;
  }

  gl_FragColor = vec4(l, r, 0., 0.);
}
```

The output is now smoother, our simple timbre sound much better... we will see next how we can play this timbre with a MIDI keyboard.

6.6 Square and triangle waveform

We can now produce many basic/common waveforms such as a square wave or triangle wave which has only odd partials:

```
void main () {
  float l = 0.;
  float r = 0.;
  float base_frequency = 440.;

  // try with a value of 0.25 to produce a waveform close to a square wave
  float attenuation_constant = 1.95;

  const float harmonics = 8.;

  // notice how we are stepping through odd harmonics
  for (float i = 1.; i < harmonics; i += 2.) {
    float a = 1. / pow(i, attenuation_constant);

    l += fline(base_frequency * i) * a;
    r += fline(base_frequency * i) * a;
  }

  gl_FragColor = vec4(l, r, 0., 0.);
}
```

Both the square wave and triangle wave have odd partials but a triangle wave has gradually much weaker high harmonics.

6.7 Simulating pulse-width

Pulse-width is a feature commonly found in analog synthesizer, by changing the width of a pulse it is possible to change the harmonic content, it goes from a square wave at 50% and approach the sound of a crisper sawtooth wave near 0%.

```
// #define statement cannot begin on the first line in Fragment
#define pi 3.141592653589

void main () {
  float l = 0.;
  float r = 0.;
  float base_frequency = 440.;

  // will produce a waveform akin to a square wave
  // try with 0.1 for a waveform akin to a sawtooth wave
  float pulse_width = 0.5;

  const float harmonics = 16.;

  for (float i = 1.; i < harmonics; i += 1.) {
    // plot this function with a graph ploting software
    // for details
    float pm = abs(cos((i * pi - pi) * pulse_width));

    l += fline(base_frequency * i) * pm;
    r += fline(base_frequency * i) * pm;
  }

  gl_FragColor = vec4(l, r, 0., 0.);
}
```

The advantage of a pulse-width in analog synthesizers is that the width can be modulated to produce string-like patches and is an easy way to change the harmonic content.

Within Fragment, the pulse-width is just one of the unlimited way to change the harmonic content of a timbre.

6.8 Playing with a MIDI keyboard

You must have a WebMIDI supported browser (such as Chrome or Opera) to use MIDI messages with Fragment.

Once your MIDI keyboard is plugged in, you can go in the Fragment MIDI devices dialog and enable it.

The MIDI data will now be available in a pre-defined **vec4 array** named **keyboard**, the array length is the actual polyphony capability of Fragment, the array contain a series of vec4 items, a vec4 item contain:

- the note-on frequency

- the note velocity
- the elapsed time since the key was pressed
- the MIDI channel

Important note: There is no notions of "note-off" in Fragment, a note-off is simply an item filled with 0, Fragment will add note-on in order and will reorder the array as necessary, this mean that you can loop over all items and break the loop if an item component is equal to 0, it is possible to detect note-off events by using the previous frame (aka feedback feature).

Here is how you can use a MIDI keyboard:

```
void main () {
  float l = 0.;
  float r = 0.;

  float attenuation_constant = 1.95;

  const float harmonics = 8.;

  // 8 notes polyphony
  // this can be increased up to the maximal polyphonic capabilities
  // of Fragment for your platform
  for (int k = 0; k < 8; k += 1) {
    // we get the note-on data
    vec4 data = keyboard[k];

    float kfrq = data.x; // frequency
    float kvel = data.y; // velocity
    float ktim = data.z; // elpased time
    float kchn = data.w; // channel

    // we quit as soon as there is no more note-on data
    // this is optional but it might help with performances
    // so we don't compute harmonics uselessly
    if (kfrq == 0.) {
     break;
    }

    for (float i = 1.; i < harmonics; i += 1.) {
      float a = 1. / pow(i, attenuation_constant);

      // we apply the velocity
      l += fline(kfrq * i) * a * kvel;
      r += fline(kfrq * i) * a * kvel;
    }
  }

  // notice how the blue channel is used
  // this help to get a better visual feedback of the harmonics
  gl_FragColor = vec4(l, r, l * 64., 0.);
}
```

You can now play something with your MIDI keyboard and hear your simple timbre!

You might find the loop kind of weird, why we don't just loop over all available note-on messages? This is because the GLSL spec. for the version that is used by all browsers is quite limited when it come to loop features... for example, it does not allow dynamicals indexes.

6.9 Simple AR-like envelope

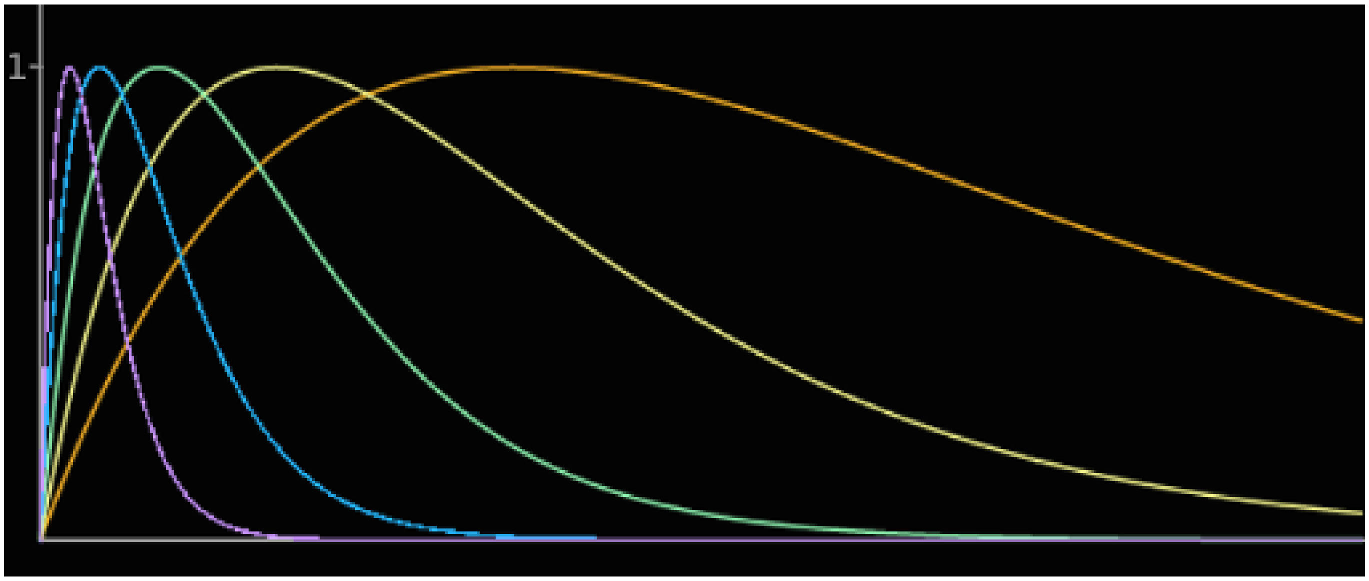
Our timbre is nice but still too simple, we can fix that by applying an envelope, a mathematical function which will determine the timbre behavior by taking into account the elapsed time when we hit the keys of our MIDI keyboard.

Fragment allow any envelopes to be done by mathematically defining them, one of the simplest envelope is AR (Attack and Release):

```
// a simple function which define
// smooth AR-like envelopes, fast attack and slow decays
// plot it or see the details of this function below
// it's maximum, which is 1.0, happens at exactly x = 1 / k
// use k to control the stretching of the function
float impulse(float k, float x) {
  float h = k * x;
  return h * exp(1.0 - h);
}

void main () {
  float l = 0.;
  float r = 0.;

  float attenuation_constant = 1.95;

  const float harmonics = 8.;

  for (int k = 0; k < 8; k += 1) {
    vec4 data = keyboard[k];

    float kfrq = data.x;
    float kvel = data.y;
    float ktim = data.z;
    float kchn = data.w;

    if (kfrq == 0.) {
     break;
    }

    // we apply the envelope, this envelope has a fast attack
    // of 62.5ms and a release of 555ms
    float env_attack = ktim * 16.;
    float env = kvel * impulse(1., env_attack);

    for (float i = 1.; i < harmonics; i += 1.) {
      float a = 1. / pow(i, attenuation_constant);

      l += fline(kfrq * i) * a * env;
      r += fline(kfrq * i) * a * env;
    }
  }

  gl_FragColor = vec4(l, r, l + r, 0.);
}
```

You can change the parameters of the envelope to apply other kinds of envelopes.

Here is a good preview of the envelope models that the **impulse** function generate:



`(images/impulse.png)`
The sound is still not very interesting yet but now that we know how to generate the envelope, we can do many things such as applying a different envelope to each harmonics:

```
float impulse(float k, float x) {
  float h = k * x;
  return h * exp(1.0 - h);
}

void main () {
  float l = 0.;
  float r = 0.;

  float attenuation_constant = 1.95;

  // we will now increase the amount of harmonics
  const float harmonics = 16.;

  for (int k = 0; k < 8; k += 1) {
    vec4 data = keyboard[k];

    float kfrq = data.x;
    float kvel = data.y;
    float ktim = data.z;
    float kchn = data.w;

    if (kfrq == 0.) {
     break;
    }

    float env_attack = ktim * 16.;

    for (float i = 1.; i < harmonics; i += 1.) {
      float a = 1. / pow(i, attenuation_constant);

      // we stretch the envelope per harmonics
      float env_stretch = i / harmonics;
      float env = kvel * impulse(env_stretch, env_attack);

      l += fline(kfrq * i) * a * env;
      r += fline(kfrq * i) * a * env;
    }
  }

  gl_FragColor = vec4(l, r, l + r, 0.);
}
```

The timbre produced by this code is very pleasant, all high frequencies harmonics decay very fast while the fundamental decay slowly, you can hear multiple versions of this by varying parameters such as the attack time or the attenuation constant etc., a very fast attack time sound very pleasant now.

6.10 Morphing between parameters

We will now look into getting a better timbre by morphing parameters such as the attenuation or frequencies, due to the nature of Fragment, almost everything can be morphed easily by using mathematical functions:

```
float impulse(float k, float x) {
  float h = k * x;
  return h * exp(1.0 - h);
}

void main () {
  float l = 0.;
  float r = 0.;

  float attenuation_constant = 1.95;

  const float harmonics = 16.;

  for (int k = 0; k < 8; k += 1) {
    vec4 data = keyboard[k];

    float kfrq = data.x;
    float kvel = data.y;
    float ktim = data.z;
    float kchn = data.w;

    if (kfrq == 0.) {
     break;
    }

    float env_attack = ktim * 16.;

    // we create a mixing function
    float mix_f = 0.5 * (1. + sin(ktim * 20.));

    for (float i = 1.; i < harmonics; i += 1.) {
      float a = 1. / pow(i, attenuation_constant);

      // we setup another attenuation value which
      // is a bit higher than the previous one
      float a2 = 1. / pow(i, attenuation_constant * 1.3);

      float env_stretch = i / harmonics;
      float env = kvel * impulse(env_stretch, env_attack);

      // we now mix our attenuations so that
      // higher harmonics oscillate between both
      // kind of attenuations smoothly
      a = mix(a, a2, mix_f);

      l += fline(kfrq * i) * a * env;
      r += fline(kfrq * i) * a * env;
    }
  }

  gl_FragColor = vec4(l, r, l + r, 0.);
}
```

The timbre is much more pleasant, especially if you play some chords.

This is just an example but you can now imagine the amount of possibilities, very complex timbres can be created easily by applying custom functions and mixing between several parameters!

For example, we could do a tremolo or vibrato effect very easily by modulating our harmonics or frequencies quickly!

Fragment offer a limitless landscape for any creative minds willing to experiment.

6.11 ADSR envelope

We will now replace our AR envelope by the more complex ADSR envelope, ADSR mean Attack, Decay, Sustain, Release, this is a classic and is the most commonly used envelope, this will enhance our possibilities.

```
  float adsr(float t, vec4 v, float s) {
    v.xyw = max(vec3(2.2e-05), v.xyw);
    // attack term
    float ta = t/v.x;
    // decay / sustain amplitude term
    float td = max(s, 1.0-(t-v.x)*(1.0-s)/v.y);
    // length / release term
    float tr = (1.0 - max(0.0,t-(v.x+v.y+v.z))/v.w);
    return max(0.0, min(ta, tr*td));
  }

  void main () {
    float l = 0.;
    float r = 0.;

    float attenuation_constant = 1.95;

    const float harmonics = 16.;

    for (int k = 0; k < 8; k += 1) {
      vec4 data = keyboard[k];

      float kfrq = data.x;
      float kvel = data.y;
      float ktim = data.z;
      float kchn = data.w;

      if (kfrq == 0.) {
       break;
      }

      float mix_f = 0.5 * (1. + sin(ktim * 2.));

      for (float i = 1.; i < harmonics; i += 1.) {
        float a = 1. / pow(i, attenuation_constant * 1.1);

        float a2 = 1. / pow(i, attenuation_constant * 1.3);

        // we now replace our old impulse function by the adsr function
        // which take three arguments
        // time, a vec4 with ADSR parameters and the value to decay to
        float attack = 0.25; // 250ms attack
        float decay = 1.; // 1 sec decay (to 0.25, see decay_amp)
        float sustain = 0.; // no sustain
        float release = 0.25; // 250ms release
        float decay_amp = 0.25;
        float env = kvel * adsr(ktim, vec4(attack, decay, sustain, release),
decay_amp);

        a = mix(a, a2, mix_f);

        l += fline(kfrq * i) * a * env;
        r += fline(kfrq * i) * a * env;
      }
    }

    gl_FragColor = vec4(l, r, l + r, 0.);
  }
```

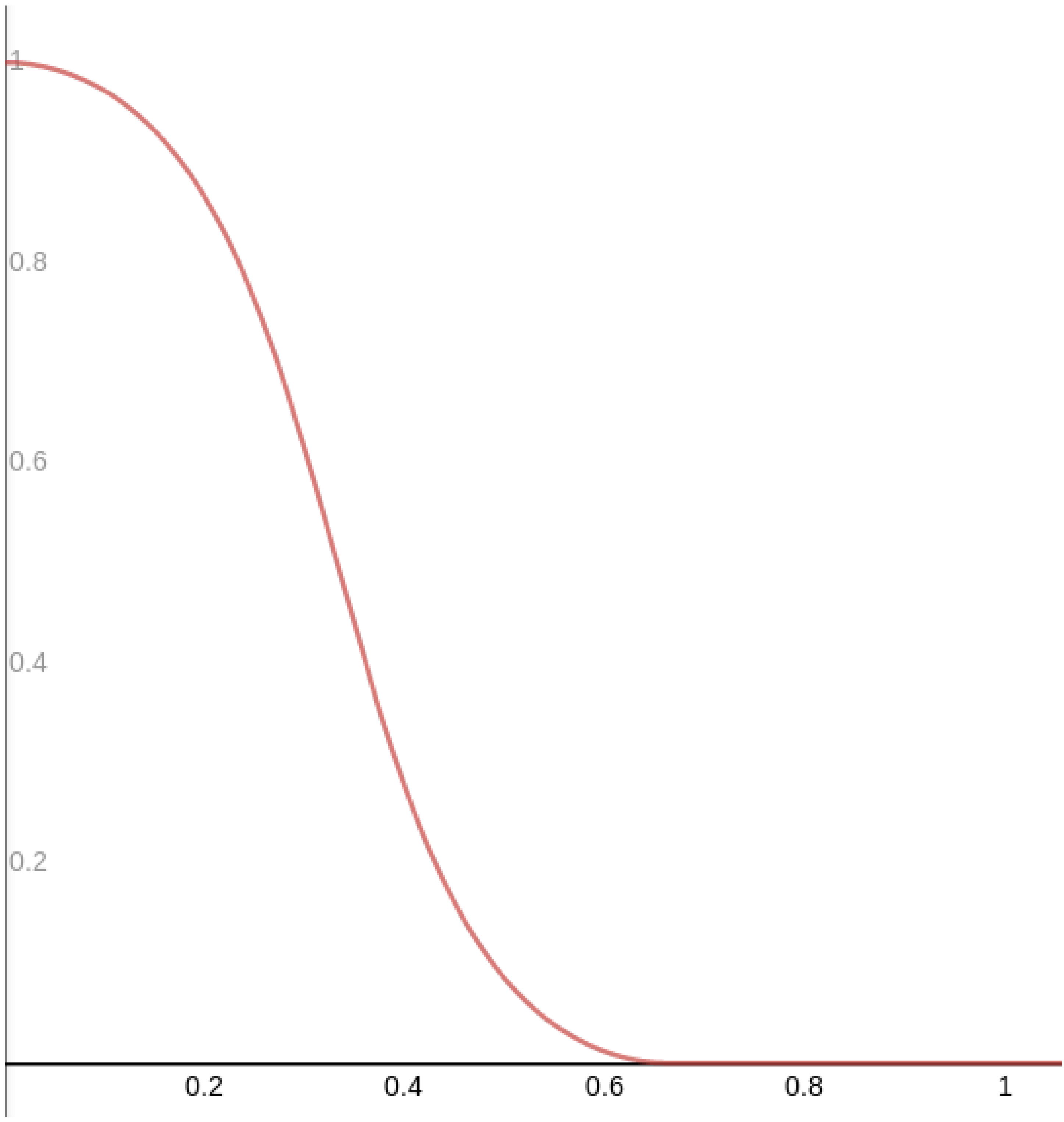
6.12 Filters

Filters play an important role in sound design, we will see in this section how to apply more powerful and flexible filters with parameters that we can modulate, we will especially recreate generic filters such as low-pass, high-pass etc.

6.12.1 Low-pass filter (LPF)

A low-pass filter is a filter that passes signals with a frequency lower than a certain cutoff frequency and attenuates signals with frequencies higher than the cutoff frequency.

We will use the low-pass filter as a prototype filter, which mean that we will build all the other filters from this one.



`(images/lpf.png)`

The graph of the lpf(x, 0., 1.5) function

The attenuation function that we have used so far is a kind of simple low-pass filter.

```
float adsr(float t, vec4 v, float s) {
  v.xyw = max(vec3(2.2e-05), v.xyw);
  float ta = t/v.x;
  float td = max(s, 1.0-(t-v.x)*(1.0-s)/v.y);
  float tr = (1.0 - max(0.0,t-(v.x+v.y+v.z))/v.w);
  return max(0.0, min(ta, tr*td));
}

// low-pass filter
// x is the current normalized position
// c is the cutoff position
// s is the slope
float lpf(float x, float c, float s) {
  return .5 + .5 *(atan(s*cos(3.141592653 * min(max((x-c) * s,0.),1.)))/atan(s));
}

void main () {
  // we will need the current normalized coordinates
  vec2 uv = gl_FragCoord.xy / resolution.xy;

  float l = 0.;
  float r = 0.;

  const float harmonics = 16.;

  for (int k = 0; k < 8; k += 1) {
    vec4 data = keyboard[k];

    float kfrq = data.x;
    float kvel = data.y;
    float ktim = data.z;
    float kchn = data.w;

    if (kfrq == 0.) {
     break;
    }

    // the htoy function take a frequency as argument
    // and return its position on the canvas
    // we then normalize the position
    // the base frequency of our note will be directly used as the cutoff frequency
    float cutoff = htoy(kfrq) / resolution.y;
    for (float i = 1.; i < harmonics; i += 1.) {
      // our slope factor, high frequencies will be attenuated better
      float slope = 3.;
      float f = lpf(uv.y, cutoff, slope);

      float attack = 0.05;
      float decay = 0.6;
      float sustain = 0.;
      float release = 0.25;
      float dec_amp = 0.;
      float env = kvel * adsr(ktim, vec4(attack, decay, sustain, release), dec_amp);

      l += fline(kfrq * i) * f * env;
      r += fline(kfrq * i) * f * env;
    }
  }

  gl_FragColor = vec4(l, r, 0., 0.);
}
```

We can now modulate the filter parameters to change the timbre, we will vary the slope factor of our filter:

```
float adsr(float t, vec4 v, float s) {
  v.xyw = max(vec3(2.2e-05), v.xyw);
  float ta = t/v.x;
  float td = max(s, 1.0-(t-v.x)*(1.0-s)/v.y);
  float tr = (1.0 - max(0.0,t-(v.x+v.y+v.z))/v.w);
  return max(0.0, min(ta, tr*td));
}

float lpf(float x, float c, float s) {
  return .5 + .5 *(atan(s*cos(3.141592653 * min(max((x-c) * s,0.),1.)))/atan(s));
}

void main () {
  vec2 uv = gl_FragCoord.xy / resolution.xy;

  float l = 0.;
  float r = 0.;

  const float harmonics = 16.;

  for (int k = 0; k < 8; k += 1) {
    vec4 data = keyboard[k];

    float kfrq = data.x;
    float kvel = data.y;
    float ktim = data.z;
    float kchn = data.w;

    if (kfrq == 0.) {
     break;
    }

    float cutoff = htoy(kfrq) / resolution.y;

    // we vary the slope a bit
    float slope = 3. + sin(ktim * 4.);

    for (float i = 1.; i < harmonics; i += 1.) {
      float f = lpf(uv.y, cutoff, slope);

      float attack = 0.05;
      float decay = 0.6;
      float sustain = 0.;
      float release = 0.25;
      float dec_amp = 0.;
      float env = kvel * adsr(ktim, vec4(attack, decay, sustain, release), dec_amp);

      l += fline(kfrq * i) * f * env;
      r += fline(kfrq * i) * f * env;
    }
  }

  gl_FragColor = vec4(l, r, 0., 0.);
}
```

The timbre is a bit softer, if you increase the modulation frequency, some nice sounds can be made.

6.12.2 High-pass filter (HPF), Band-pass filter and Band-reject filter

Now that we have made a low-pass filter, we can build all the other filters easily:

```
float adsr(float t, vec4 v, float s) {
  v.xyw = max(vec3(2.2e-05), v.xyw);
  float ta = t/v.x;
  float td = max(s, 1.0-(t-v.x)*(1.0-s)/v.y);
  float tr = (1.0 - max(0.0,t-(v.x+v.y+v.z))/v.w);
  return max(0.0, min(ta, tr*td));
}

float lpf(float x, float c, float s) {
  return .5 + .5 *(atan(s*cos(3.141592653 * min(max((x-c) * s,0.),1.)))/atan(s));
}

// the high pass filter is just an inversed LPF
float hpf(float x, float c, float s) {
 return lpf(1. - x, 1. - c, s);
}

// a bpf is obtained by combining a lpf and hpf
float bpf(float x, float c, float s) {
  return lpf(x, c, s) * hpf(x, c, s);
}

// band-reject
float brf(float x, float c, float s) {
  return (1. - lpf(x, c, s)) + (1. - hpf(x, c, s));
}

void main () {
  vec2 uv = gl_FragCoord.xy / resolution.xy;

  float l = 0.;
  float r = 0.;

  const float harmonics = 16.;

  for (int k = 0; k < 8; k += 1) {
    vec4 data = keyboard[k];

    float kfrq = data.x;
    float kvel = data.y;
    float ktim = data.z;
    float kchn = data.w;

    if (kfrq == 0.) {
     break;
    }

    float cutoff = htoy(kfrq) / resolution.y;

    float slope = 3. + sin(ktim * 4.);

    // we compute our filters
    // note that we modulate the cutoff parameter
    // to show how the filters behave
    float fhp = hpf(uv.y, cutoff + sin(ktim * 1.25) / 2., slope);
    float fbp = bpf(uv.y, cutoff + sin((ktim - 1.) * 1.25) / 2., slope * 1.25);
    float fbr = brf(uv.y, cutoff + sin((ktim - 2.) * 1.25) / 2., slope * 1.5);

    float attack = 0.5;
    float decay = 0.5;
    float sustain = 0.;
    float release = 0.25;
```

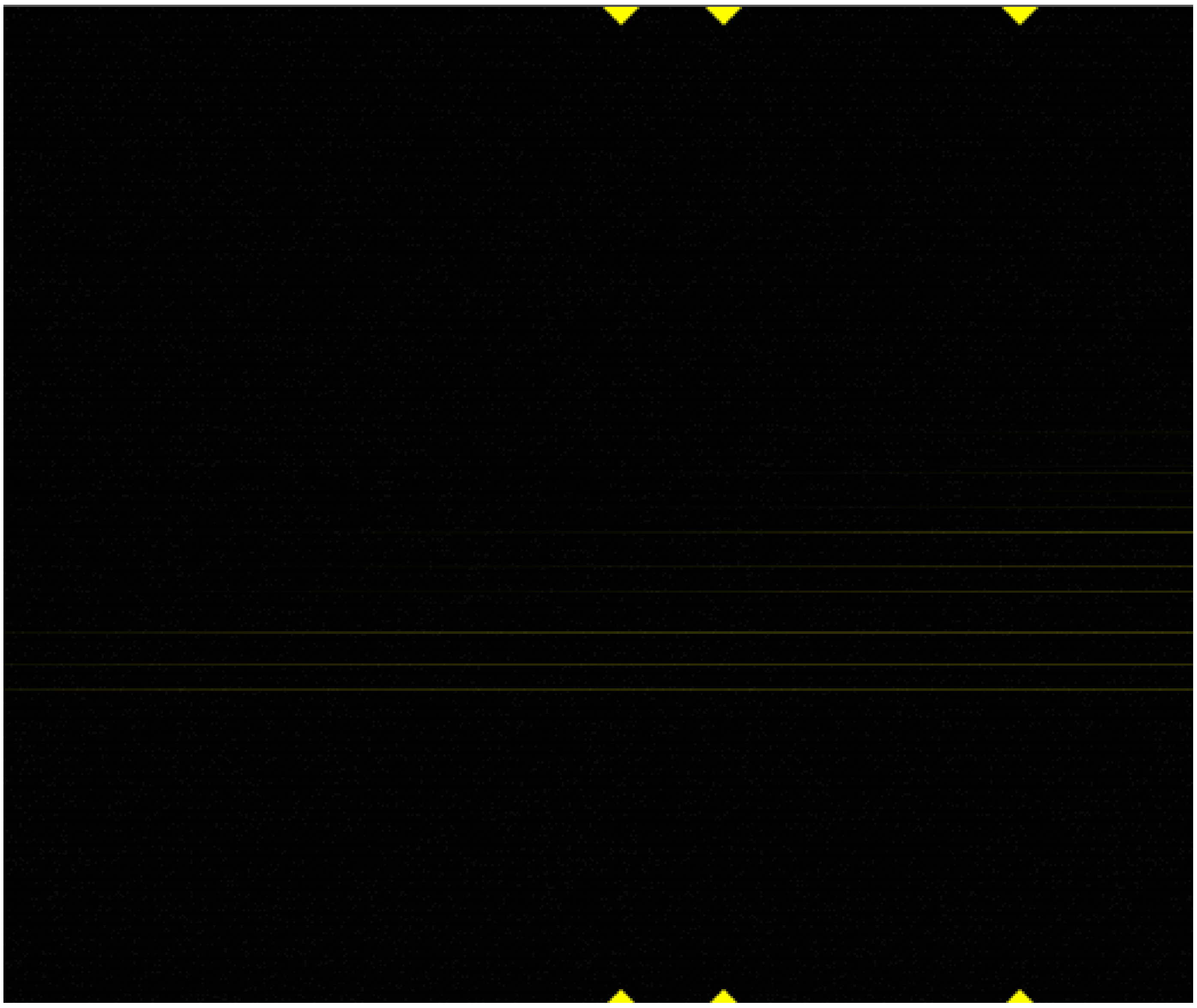
The given functions are not set in stone, you can change them, those functions have some drawbacks actually, notably the fact that they don't have a resonance parameter and that the filters cutoff/slope need to be adjusted for HPF/BPF/BRF because the attenuation is not per octaves, we will see next some other functions which will adress these issues.

6.13 Timbral possibilities

A neat thing that can be done with Fragment is modulating the parameters you want with the current horizontal or vertical position, the canvas will then show you many timbral possibilities that you can chose to listen to by adding and positioning slices on the canvas, you can also modulate the horizontal position to move between timbral possibilities.

By adding many slices at different positions, you can build very complex sounds and explore without having to write code.

```
float adsr(float t, vec4 v, float s) {
  v.xyw = max(vec3(2.2e-05), v.xyw);
  float ta = t/v.x;
  float td = max(s, 1.0-(t-v.x)*(1.0-s)/v.y);
  float tr = (1.0 - max(0.0,t-(v.x+v.y+v.z))/v.w);
  return max(0.0, min(ta, tr*td));
}

float lpf(float x, float c, float s) {
  return .5 + .5 *(atan(s*cos(3.141592653 * min(max((x-c) * s,0.),1.)))/atan(s));
}

void main () {
  vec2 uv = gl_FragCoord.xy / resolution.xy;

  float l = 0.;
  float r = 0.;

  const float harmonics = 16.;

  for (int k = 0; k < 8; k += 1) {
    vec4 data = keyboard[k];

    float kfrq = data.x;
    float kvel = data.y;
    float ktim = data.z;
    float kchn = data.w;

    if (kfrq == 0.) {
     break;
    }

    float cutoff = htoy(kfrq) / resolution.y;
    float slope = 3. + sin(ktim * 16.) / 4.;

    for (float i = 1.; i < harmonics; i += 1.) {
      // notice how we multiply the cutoff by the horizontal position
      float f = lpf(uv.y, cutoff * uv.x, slope);

      float attack = 0.05;
      float decay = 0.6;
      float sustain = 0.;
      float release = 0.25;
      float dec_amp = 0.;
      float env = kvel * adsr(ktim, vec4(attack, decay, sustain, release), dec_amp);

      l += fline(kfrq * i) * f * env;
      r += fline(kfrq * i) * f * env;
    }
  }

  gl_FragColor = vec4(l, r, 0., 0.);
}
```



(images/possibilities.png)

Timbral possibilities along the X axis related to a LPF cutoff parameter

6.14 Delay effect

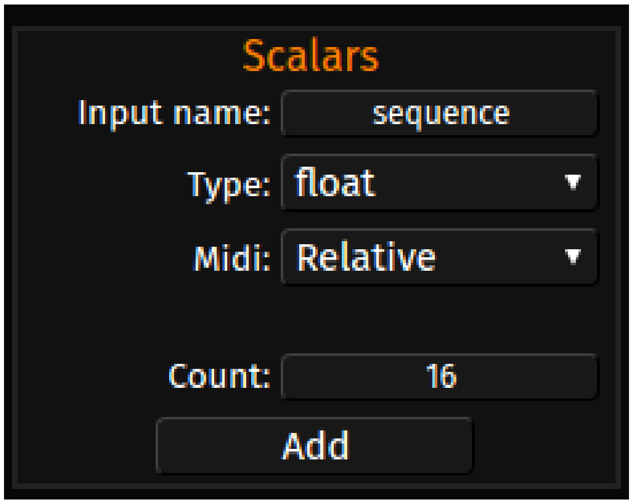
Now that we have all the essential components of any modern synthesizers, we can do some more complex things like effects, arpgeggio, sequences, filters etc.

```
Coming soon...
```

6.15 Sequence

We will see in this section how to do a 16 or N steps sequences with Fragment, we will use the uniforms dialog to define an array of values which will hold the frequency data of our sequence.

You can do that by opening the uniforms dialog and then adding an array of uniforms named **sequence** as pictured below:



Input name: sequence
Type: float
Midi: Relative
Count: 16

(`images/sequence.png`)
Many values should appear now in the uniforms dialog, you can change the frequency value for each steps with your keyboard, mouse or by assigning a MIDI control to each steps.

Here is the code to play a sequence automatically:

```
// we define the PI constant
// note: the "#define" keyword cannot start on the very first line
#define pi 3.141592653

float impulse(float k, float x) {
    float h = k * x;
    return h * exp(1.0 - h);
}

void main () {
    float l = 0.;
    float r = 0.;
    float base_frequency = 440.;

    // we just redefine globalTime for convenience
    // globalTime is a pre-defined Fragment uniform
    // which hold the time since Fragment started to play
    // you can "rewind" it by clicking on the corresponding toolbar button
    float time = globalTime;

    // increase the number of harmonics so we get a richer sound
    const float harmonics = 16.;

    // we will override "keyboard" because GLSL
    // does not permit us to assign by index dynamically
    // we will have 8 notes polyphony (copied from "keyboard" + the sequence)
    vec4 notes[9];

    // our sequence have 16 steps
    float seq_length = 16.;

    // we use the "mod" function with the length of the sequence
    // this play our sequence over and over, 4 notes per seconds
    int seq_index = int(floor(mod(time * 4., seq_length)));

    // this is a GLSL trick to get the current step frequency
    // we do that because "sequence[seq_index]" is not accepted
    // this is a limitation of the current WebGL spec.
    // which do not allow dynamical indexes for arrays...
    float seq_frq = 0.;
    for (int k = 0; k < 16; k += 1) {
      if (k == seq_index) {
        seq_frq = sequence[k];
      }
    }

    // now we just copy 8 notes from the "keyboard" array
    // which will allow us to play 8 notes at the same time
    // with a MIDI keyboard
    for (int k = 0; k < 8; k += 1) {
      notes[k] = keyboard[k];
    }

    // we assign the current sequence note
    // notice how we make the note duration
    // perfectly fit within its lifetime
    // it goes on, hit its peak and go down
    // before passing to the next
    notes[7] = vec4(seq_frq, 1., 0.5 * (1. + sin((pi * 1.5) + 2. * pi * time)), 0.);

    // we loop over our 9 notes
    for (int k = 0; k < 9; k += 1) {
```

You should hear a sequence of notes which you can play along with a MIDI keyboard.

An autonomous sequence is relatively easy and can also be done by using the **mod** function or by simpler means.

We strongly recommend that peoples use Fragment with a MIDI sequencer such as LMMS (`https://www.lmms.io`), IanniX (`https://www.iannix.org`), Renoise (`https://www.renoise.com/`), MusE (`http://muse-sequencer.org`) or any other MIDI capable sequencers, sequences can be done very easily within a sequencer and many effects can be added to the audio output.

6.16 Note on globalTime

The global time is one of the most important pre-defined variable in Fragment, it is a playback time that increase continually as soon as you hit the play button on the toolbar, it can only be reset or paused by using the corresponding toolbar button, the global time is a precise clock which can be used to trigger events and dynamically alter the visuals.

7 Contributing

Fragment is a free and open source software released under the term of the BSDv2 license.

Any contributions from the documentation to the synthesis engine are welcome!

The source code is available on GitHub (`https://github.com/grz0zrg/fsynth`).

FAS source code is also available on GitHub (`https://github.com/grz0zrg/fas`).

8 Links

- Fragment Synthesizer website (`https://www.fsynth.com`)
- Fragment Synthesizer forum (`https://quiet.fsynth.com`)
- Fragment source code (`https://github.com/grz0zrg/fsynth`)
- FAS source code (`https://github.com/grz0zrg/fas`)
- Facebook (`https://www.facebook.com/fsynth/`)
- YouTube (`https://www.youtube.com/channel/UC2CJFT1_ybPcTNlT6bVGOWQ`)
- Twitter (`https://twitter.com/fragmentsynth`)
- SoundCloud (`https://soundcloud.com/fsynth/`)

9 Credits

- Testing:
  - Franz Khrum
- Papers:
  - The Scientist and Engineer's Guide to Digital Signal Processing (`http://www.dspguide.com`)
  - Welsh's Synthesizer Cookbook (`http://www.synthesizer-cookbook.com`)
  - Fabrice Neyret Desmos page (`http://www-evasion.imag.fr/Membres/Fabrice.Neyret/demos/DesmosGraph/indexImages.html`)
- Ideas:

  - Virtual ANS (`http://www.warmplace.ru/soft/ans/`)
  - ShaderToy (`https://www.shadertoy.com/`)
- Tools:
  - The Anubis programming language (`http://redmine.anubis-language.com/`)
  - desmos (`https://www.desmos.com`)
  - jEdit (`http://www.jedit.org/`)
  - Inkscape (`https://www.inkscape.org`)
  - Geogebra (`https://www.geogebra.org`)
  - The GIMP (`https://www.gimp.org`)
  - KDEnlive (`https://kdenlive.org`)
  - SimpleScreenRecorder (`http://www.maartenbaert.be/simplescreenrecorder/`)
  - libflds (`http://liblfds.org`)
  - portaudio (`http://www.portaudio.com`)
  - libwebsockets (`https://libwebsockets.org`)
  - Brackets (`http://brackets.io/`)
  - fa2png (`http://fa2png.io/`)
  - FontAwesome (`http://fontawesome.io/`)
  - CodeMirror (`http://codemirror.net/`)
  - ShareDB (`https://github.com/share/sharedb/`)
  - live-server (`https://www.npmjs.com/package/live-server`)
  - Normalize (`https://necolas.github.io/normalize.css/`)
  - Skeleton (`http://getskeleton.com/`)
  - NodeJS (`https://nodejs.org/en/`)
  - NGINX (`https://www.nginx.com/`)
  - Flarum (`http://flarum.org/`)
  - pm2 (`https://github.com/Unitech/pm2`)
  - MongoDB (`https://www.mongodb.com/`)
  - Redis (`https://redis.io/`)
  - Winston (`https://github.com/winstonjs/winston`)
  - Express (`http://expressjs.com/`)
  - strong-cluster-control (`https://github.com/strongloop/strong-cluster-control`)

T*h*a*n*k you!

last update: 2017-4-19