Multiverse

Abhishek Choubey
Sofia Vallejo Budziszewski

Computational Music Creativity
Mappings and Expressivity
February 2023

The project aims to explore expressivity and mappings by taking a multimodal approach using music coupled with visuals. There are three major parts of the project. One is the musical section, second is the visual section, and the last is the touchOSXC interface which we used to control the musical and visual sections. The main windows of the three sections are shown below.

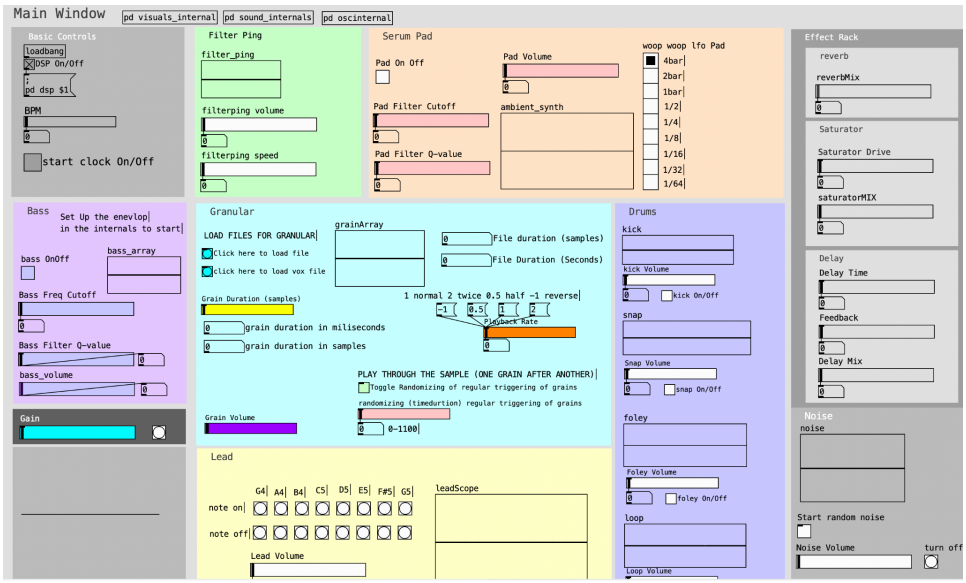

Figure 1: Musical section

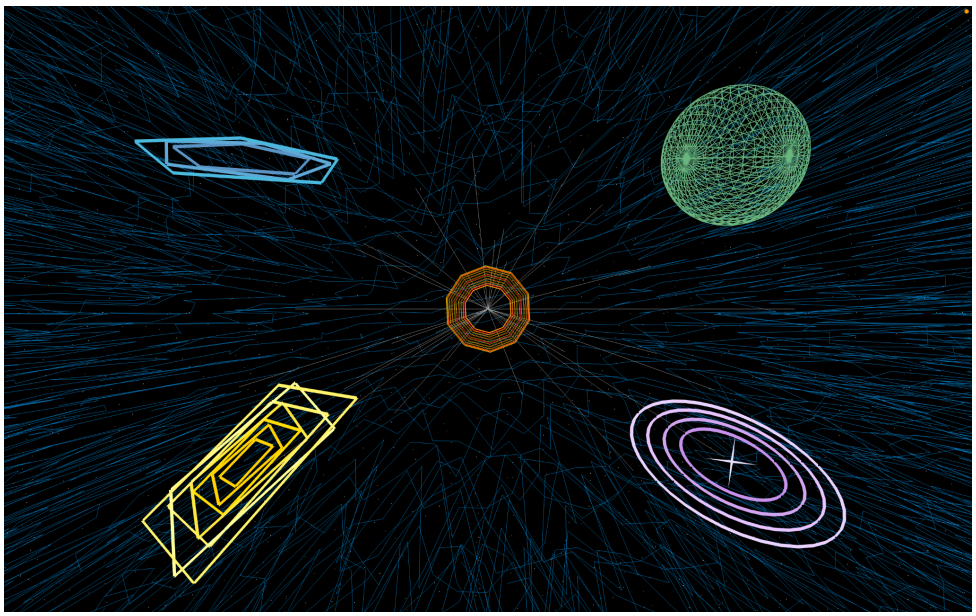

Figure 2: Visual section

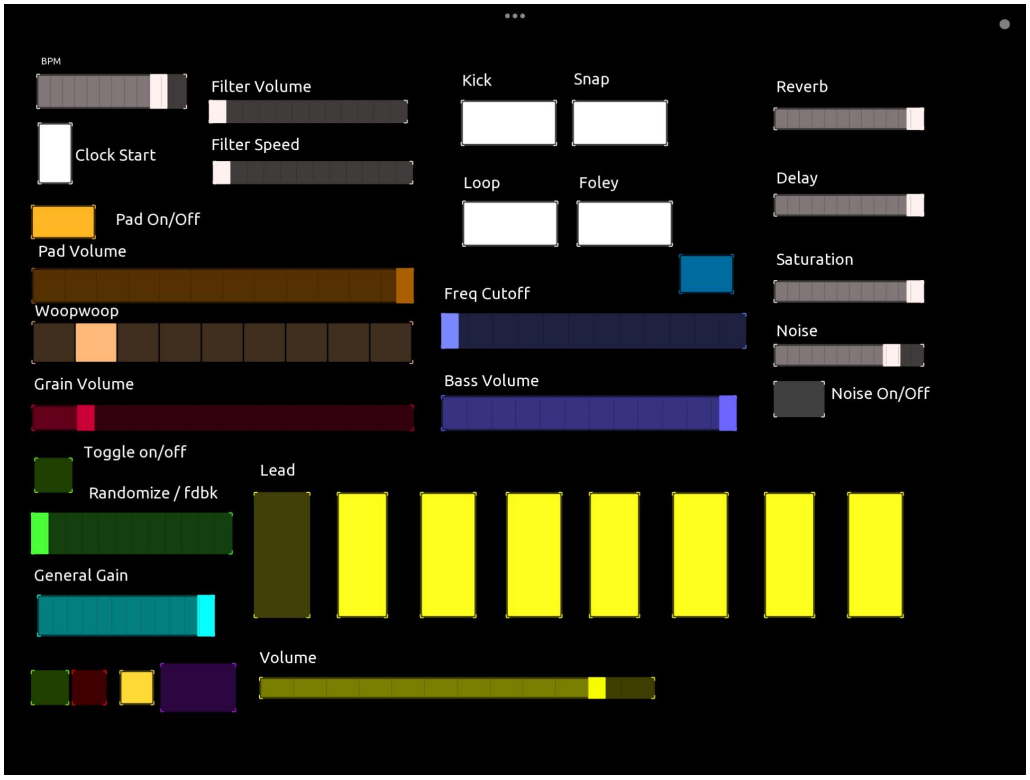

Figure 3: TouchOSC UI

1. Audio

The audio section has four major components: pads with a bassline, drums, granular vocal and melody. These are then processed using different effects: reverb saturation and delay. There are additional two sections, a simple white noise generator and filter ping. Every part except the lead melody is pre determined and plays in a loop. Only the timbre of the sound is changed using the external controls, the lead is played using the note triggers programmed in TouchOSC. The musical section is composed in the G major key. So all the choices of which chords to play and which notes to control using the touchOSC app are taken accordingly. The details of these sections and the signal flow is given below in the next part of the report. Moreover, change in audio sections triggers a change in visuals; the mapping of audio to visuals is also explained in the visuals section.

1.1 Pads and Bass

The pads are made in serum, a wavetable based synthesizer running in pure data using the vstplugin~ object. The pad is generated using four chords each play for 4 bars and then repeat. The details of the chord generation can be seen in figure 4. The serum abstraction receives 3 midi notes ON every bar and then corresponding midi note off every one second before the 4 bar time stamp. The serum abstraction and the serum preset is shown in figure 5 and 6. The serum abstraction has a parameter control section that is used to control the wavetable position of the oscillator, to control the timbre of the otherwise static chord progression. The change in wavetable position creates an effect similar to the tremolo but also plays with the timbre of the sound. The wavetable position is controlled by lfo and we in our project manipulate the rate of the lfo ranging from 4 bar to 1/64th of a note. The pads are then passed through a basic low pass filter and a high pass filter to control the energy of the pads as well as cut out frequencies that might clash with the bass and kick.

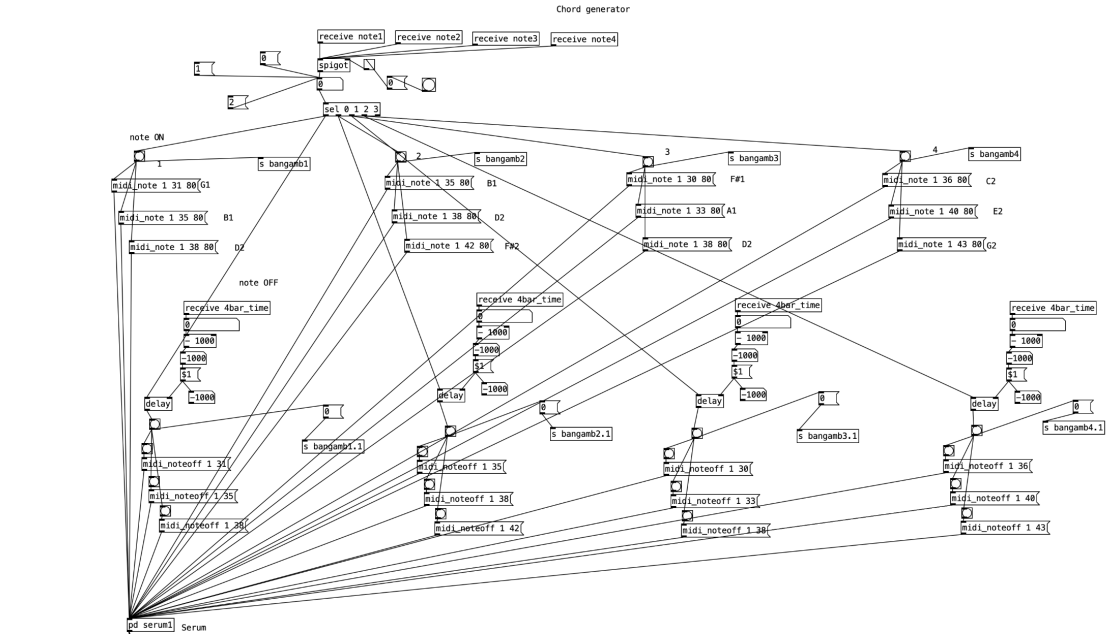

Figure 4 : chord generator (pads)

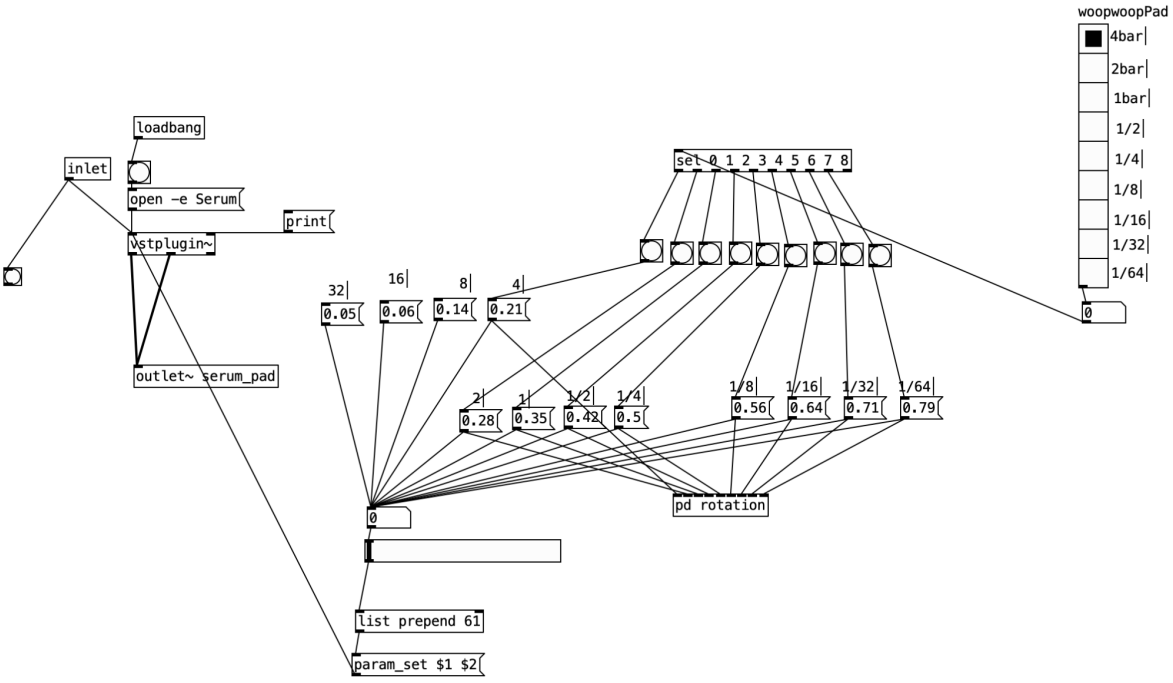

Figure 5: serum abstraction

Figure 6: serum pad preset

1.2 Melody

The lead melody is also synthesized in serum and the notes it plays are controlled using the touch OSC interface. We receive a note on and off from the touch osc interface which plays the note triggered through serum. The sound is made of a basic moog waveshap available in the serum plugin and is then controlled using an ADSR envelope, an lfo controlled low pass filter whose rate is controlled by the velocity of the midi note. So the higher the velocity the shorter the rate of the low pass filter. This again creates a tremolo effect that when passed through delay, saturation and reverb effects produces a nice rich sound. The sound compliments the pad sounds when played with bigger rate of the pads.

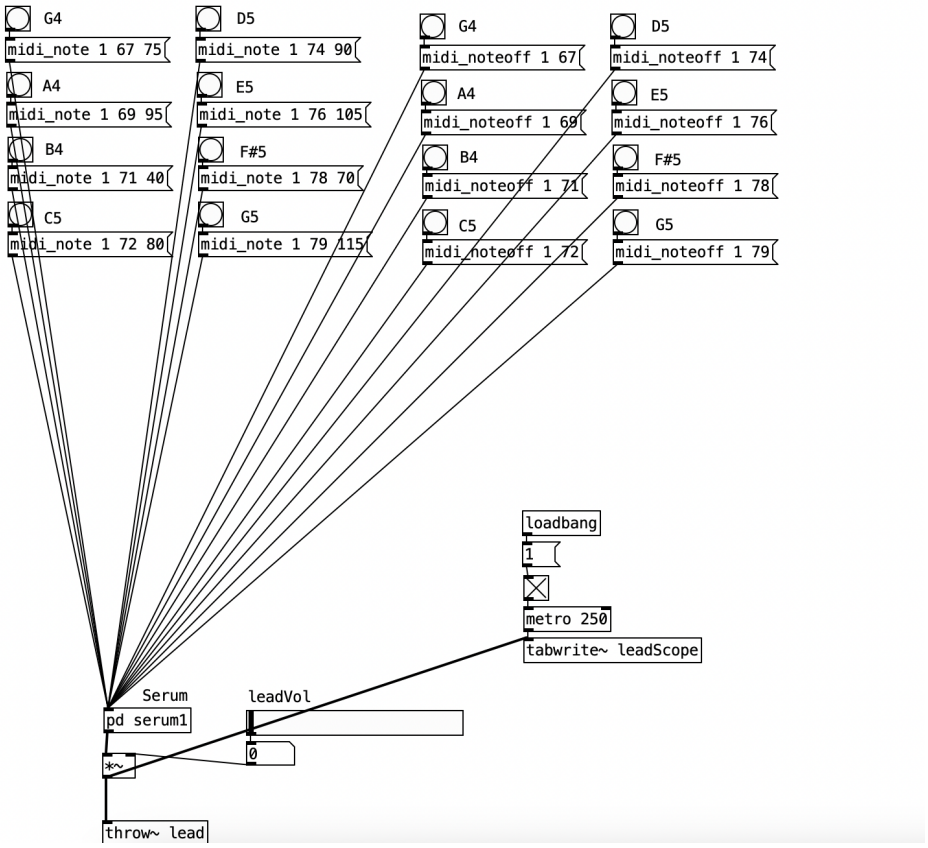

Figure 7: lead melody abstraction

Figure 8: Serum preset (lead)

1.3 Vocals

The core of the vocal section which is a granular synthesizer is borrowed from the granular synthesizer presented in the previous lab. However the controls of the synthesizers are reduced drastically. In this version the granular synthesizer only plays the vocal sample on loop but the head position of the grains is randomized and is controlled using the touchOSC UI. A vocal sample with a which is an adlib with a large reverb tail is played in a loop.

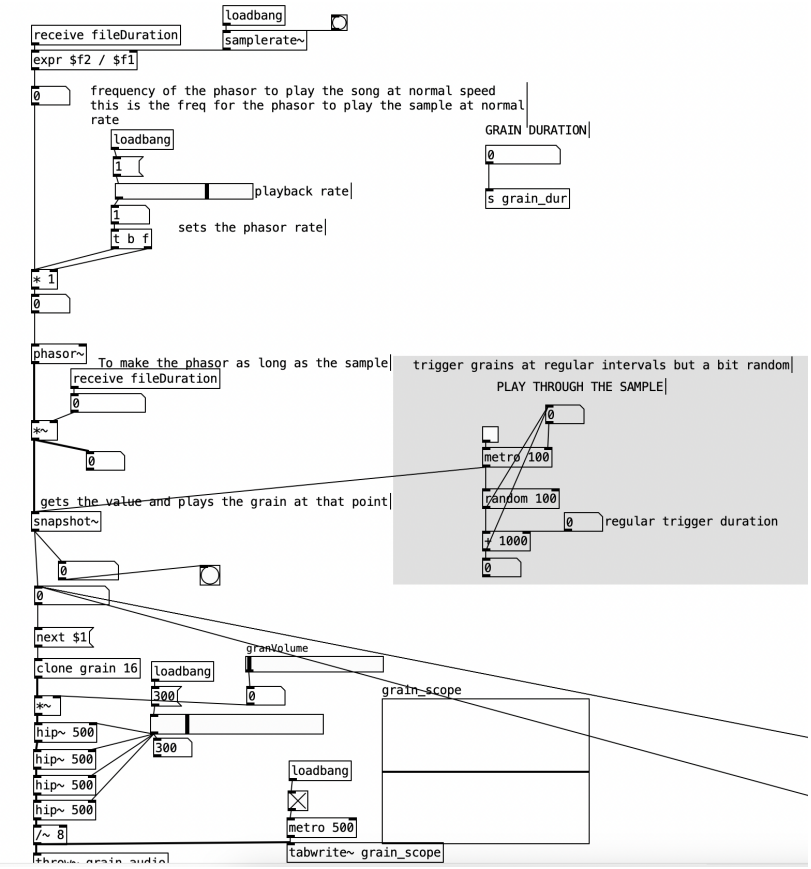

Figure 9: Granular abstraction

1.4 Drums

The drums section consists of four drum elements, a kick, snap, percussion and a short perc loop. They are all triggered in a continuous loop when turned on using a clock. The clock sends bangs every whole note, half note and quarter note and plays the drums. The kick is played every quarter note and the snap and perc are played every other half note and the loop is also played at every quarter note. The pd logic to play the loop is shown in the figure 10, all the other elements are played with the same logic except the bangs are received at times mentioned above.

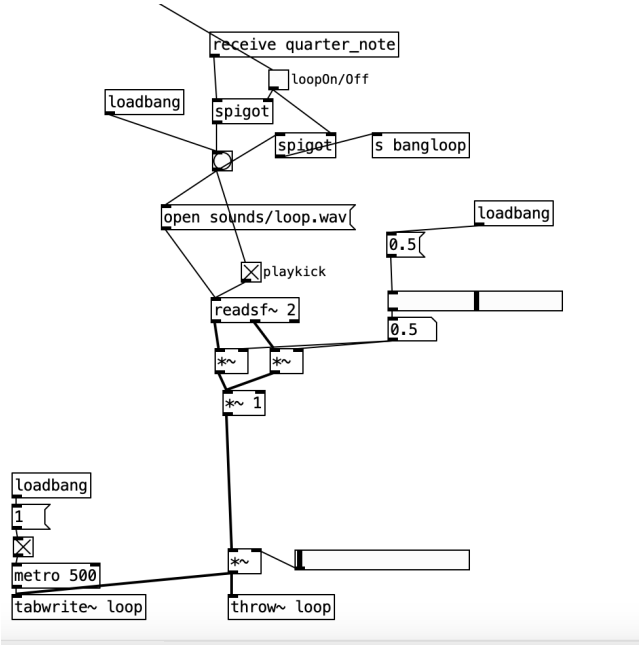

Figure 10: Loop playback

1.5 Filter ping, Effects and noise

There are two additional sections that compliment the audio produced by the above sections. The effect chain, noise and a filter ping. The effects applied are saturator, reverb and delay, additionally noise is applied to increase the chaos and energy of the system. The filter ping is an additional element that is played at the end of the piece when everything collapses.

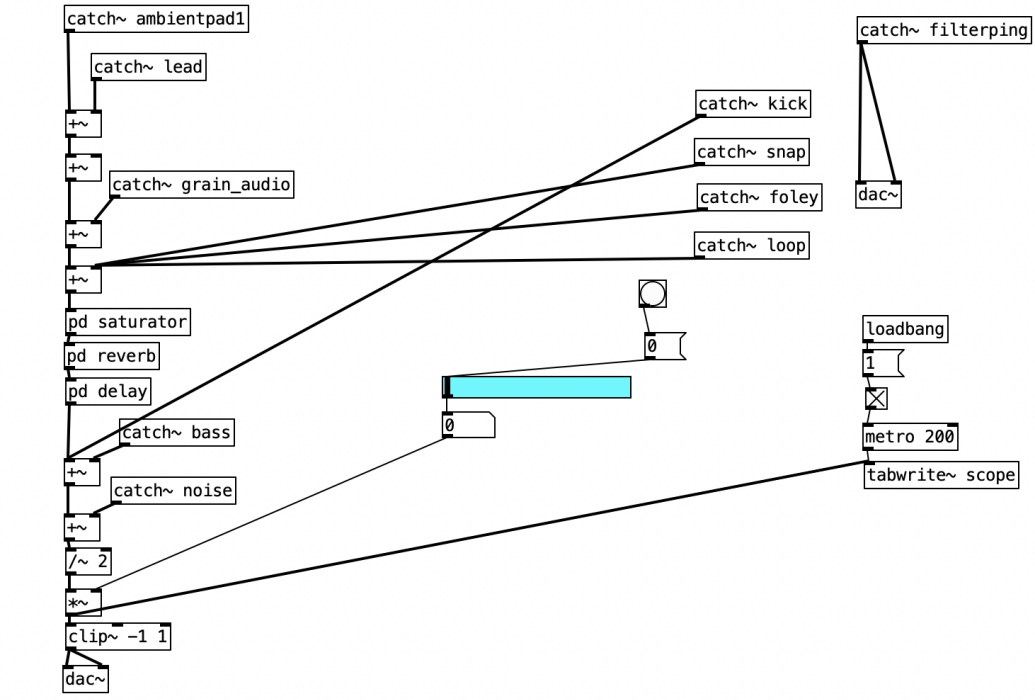

Figure 10: Noise, Effects, filter ping and signal flow

2. Visuals

The graphical patch comprises five modules, one for each instrument, and an additional module for general controls. Each instrument module incorporates a similar patch design. The X/Y button enables the user to reset all coordinates to their default values. Moreover, the main window facilitates toggling the display of individual figures on and off. The lower-left section of the interface hosts the general controls, which govern the window layout and background animations.

After exploring GEM for visual programming, we opted to use Ofelia as our solution of choice. Ofelia is a Pure Data (Pd) external that empowers artists to leverage the power of openFrameworks and Lua within a real-time visual programming environment. This framework is an ideal tool for creating stunning audiovisual artwork with great flexibility and creative freedom. For more information here is the Ofelia Github.

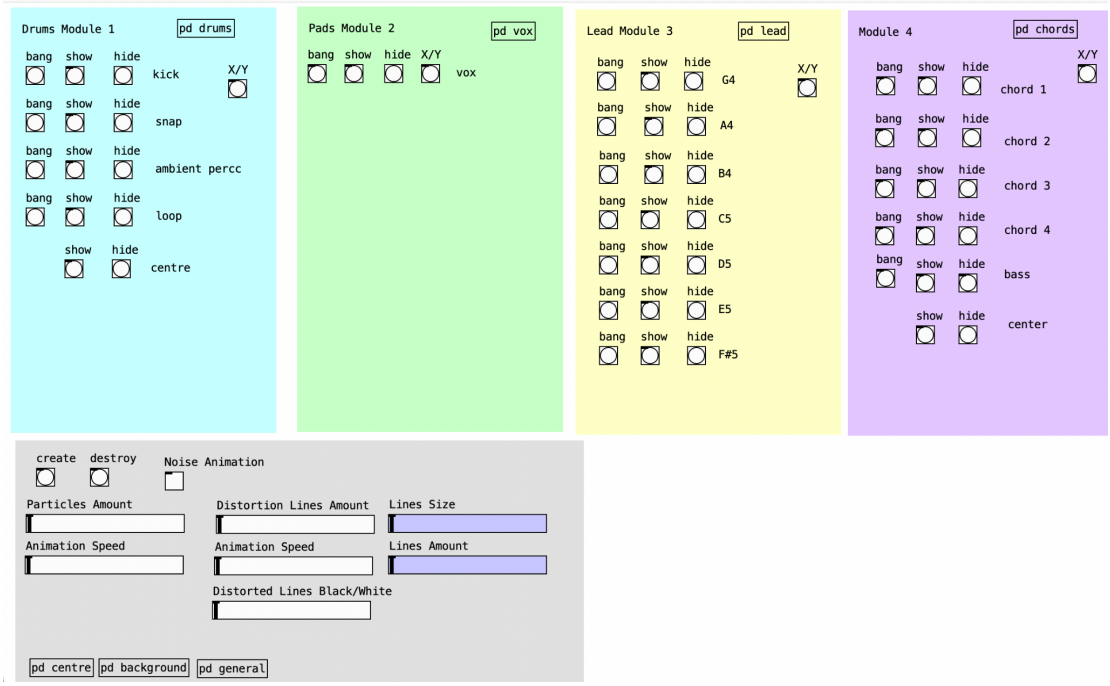

Figure 11: Visual Controls

2.1 Visual components: figures inside each module.

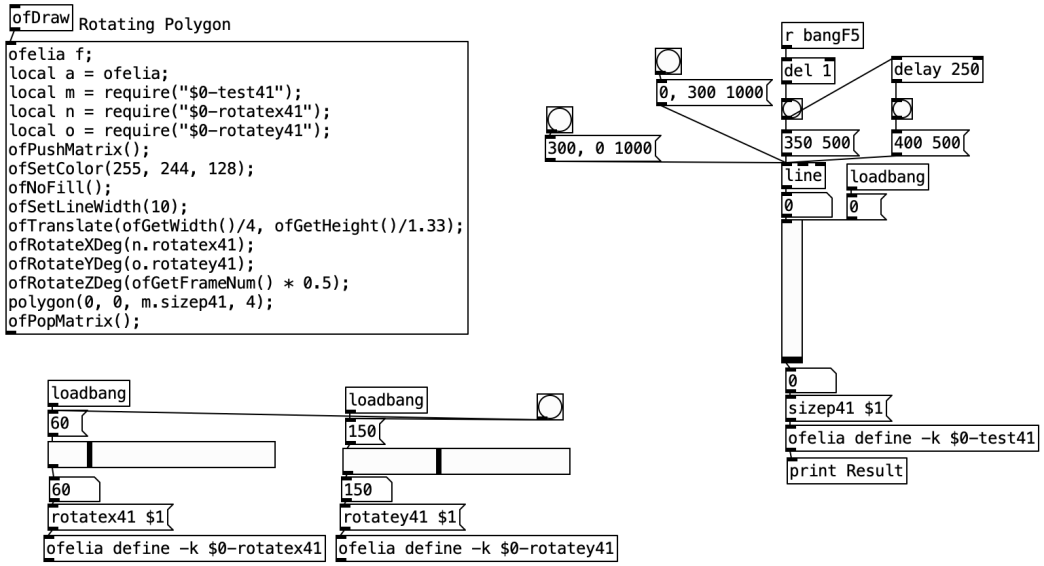

Figure 12. Inside of the Lead: F#5 module

Each module in our system is built on a consistent structure, as demonstrated in Figure 3's F#5 patch. At the top-left corner of the patch, we see the Ofelia Function object, which enables us to write custom code to create stunning visual effects. This patch leverages the Ofelia library to generate a polygon shape that rotates in 3D space, with the size and number of sides defined by variables m.sizep41 and 4, respectively. The polygon's appearance is controlled by calls to the ofSetColor(), ofNoFill(), and ofSetLineWidth() functions. It is then translated to the desired position using ofTranslate() and rendered on the screen with the polygon() function. Transformations are undone with ofPopMatrix(). The rotation angles for the X and Y axes are obtained from external files n.rotatex41 and o.rotatey41, while the Z-axis rotation angle changes dynamically based on the current frame number of the sketch.

At the top-right corner, we find the animation functions that control the visibility and size of the figure, which in this case is a square. When the F#5 note is triggered, the square shrinks from 400px to 350px in 500ms, and then, after a delay of 250ms, it expands back to its original size in another 500ms. The size of the square is regulated by the sizeep41 variable, defined within the [ofelia define -k $0-test41] object.

Finally, the bottom-left corner of the patch houses the controls for the x and y coordinates, which are particularly useful in the chords module. These controls enable us to manipulate the position of the figures on the screen and create visually dynamic and engaging artwork.

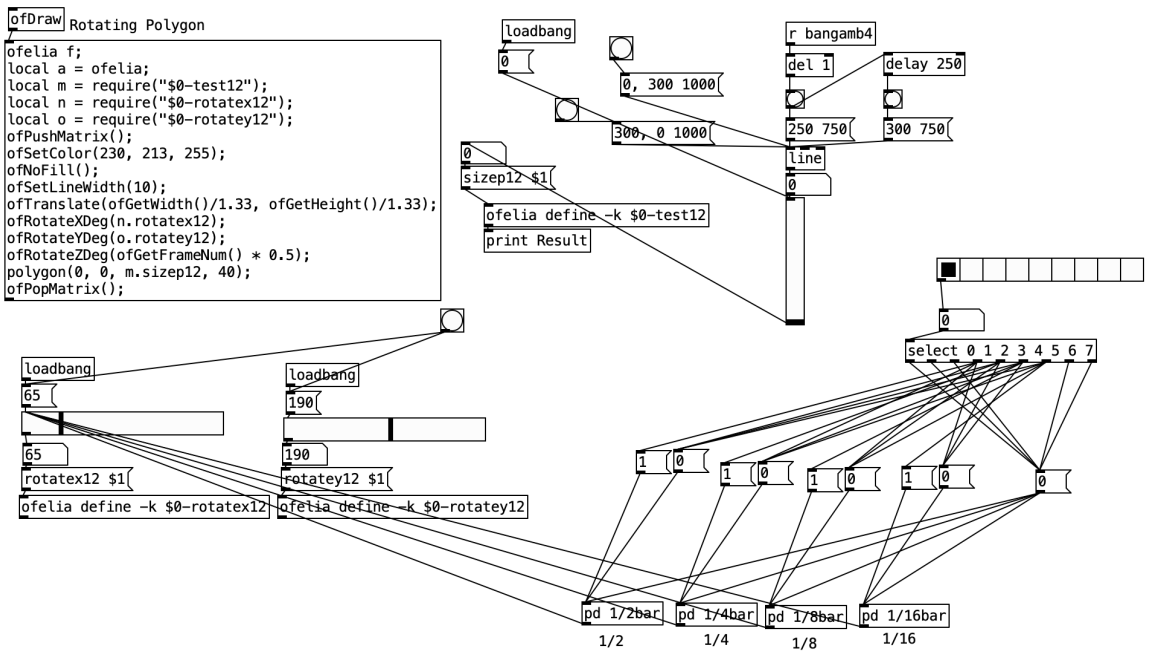

Figure 13. Usage of the coordinates controls inside the Chords: Ambient 4 module

In here, the variables rotatey12 and rotatex12 are controlled by the subpatches depending on which LFO rate is chosen, the user can choose between 4 bars, 2 bars, 1 bar, half time, quarter time, eighth time, sixteenth time, etc.

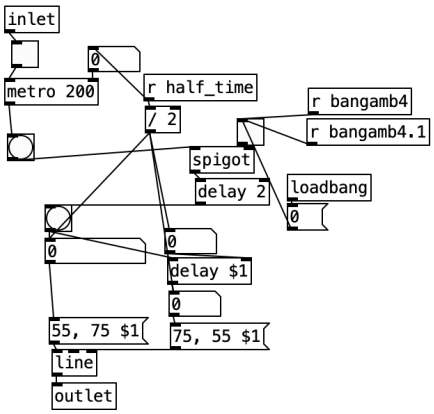

Figure 14: inside the subpatch which makes the figure vibrate with the LFO rate.

Figure 5 depicts a dynamic behavior achieved through a series of interconnected objects. When a user selects an LFO rate, a bang is triggered which sets off a metro object that sends out additional bangs exclusively when the ambience chord 4 is played. The spigot object plays a critical role here, enabling the square figure to respond only to the intended trigger. With each bang, the square figure undergoes a controlled displacement from position 75 to 55 over a duration set by the overall BPM. After a short delay of 2 ms, the square returns to its initial position, resulting in a vibrant pulsation effect that is synchronized with the LFO rate.

2.1 Visual components: background animations.

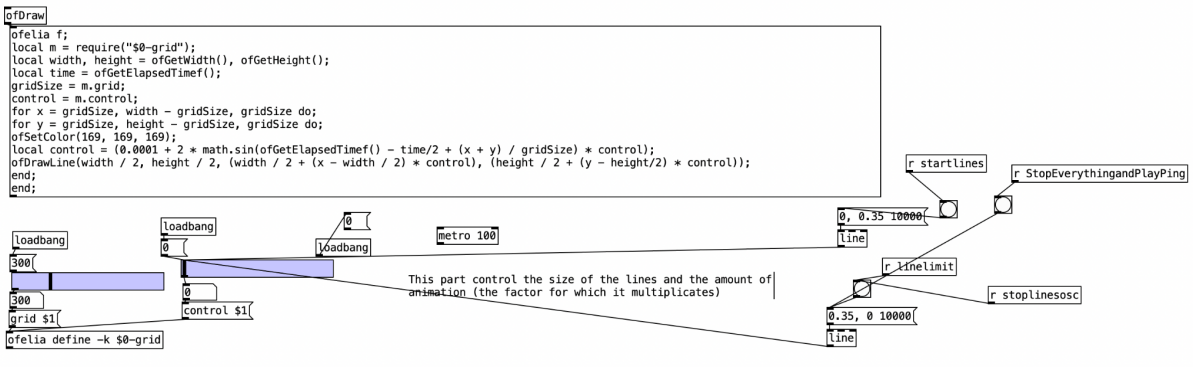

Figure 15: inside the lines patch

On the top left once again we have the ofelia function with the code which creates the lines that come out of the center. In this section of the code, the module "$0-grid" is loaded into the local variable "m". The current width and height of the display window are stored in local variables "width" and "height". A variable "time" is set to the elapsed time in seconds since the program started running. The value of "gridSize" is set to the value of "grid" from the loaded module "m", and the value of "control" is also set to the value of "control" from the same module. A for loop is then used to iterate over the grid on the x-axis with a step size of "gridSize", from "gridSize" to "width - gridSize". Within this loop, another for loop is used to iterate over the grid on the y-axis with a step size of "gridSize", from "gridSize" to "height - gridSize". To draw the lines, a shade of gray is set using ofSetColor. A variable "control" is then updated to a value that varies over time, which depends on the elapsed time, the position on the grid, and the value of "control" from the loaded module. Finally, the line is drawn using ofDrawLine from the center of the display window to a point on the grid, with the position of the endpoint influenced by the value of "control". This process is repeated for each point on the grid, resulting in a changing grid-like pattern that evolves over time.
In the bottom part of Figure 6, the size and animation speed of the lines are controlled. Specifically, the size of the lines is animated to grow from 0 to 0.35 pixels over a period of 10 seconds, and then back to 0 when a trigger is received by the "StopEverythingAndPlayPing" object.

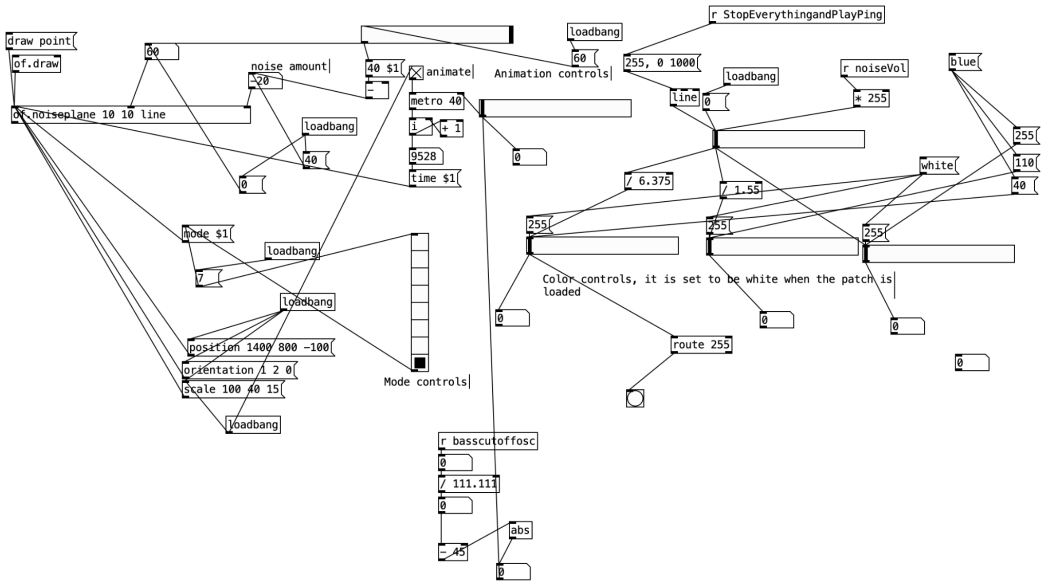

Figure 16: Inside the distorted lines and particles subpatch

In this part, the distorted blue lines are created using the [of.noiseplane] object. This is a function that generates a two-dimensional Perlin noise texture. Perlin noise is a type of gradient noise used in computer graphics and procedural content generation to create natural-looking textures, such as clouds, fire, and water. The [of.noiseplane] function takes three arguments: the width and height of the texture in pixels, and the resolution of the noise. The resolution determines the scale of the noise: smaller values produce fine-grained, detailed noise, while larger values produce larger, more abstract patterns. The function returns an image object containing the generated noise texture. Once the noise texture is generated, it can be used in various ways, such as as a displacement map to distort the vertices of a mesh, or as a height map to create terrain. It can also be used as a source of random values for procedural animation, or to add variation to other types of graphics. The noise function can also be used to generate one-dimensional and three-dimensional noise.
In the code we used for the noiseplane function, the code defines a Lua script using the Ofelia library for OpenFrameworks. It creates a 3D plane mesh and allows for various parameters to be set, such as size, number of rows and columns, draw mode, stroke weight, position, orientation, and texture mapping. The "bang" function is executed on every frame and updates the z-position of the vertices based on Perlin noise, allowing for an animated effect. The mesh is then drawn in the specified draw mode.

In Figure 7, we have demonstrated how the user has control over the animation speed, noise/lines, and color to achieve the desired effect. Specifically, we utilized these controls to make the lines appear and disappear in our project. We associated the amount of noise with the distorted lines, and the amount of reverb with the particles.

3. Open Sound Control (OSC)

We utilized OSC to enable control of the patch through a Touch OSC interface on an iPad. In the figure above, the [netreceive] object retrieves information on a specified port (port 1112 in this case). Each button and fader on the Touch OSC interface was mapped to specific parameters within the patch, such as the bass cutoff frequency and delay mix.

Due to the resource-intensive nature of the project, running both audio and video simultaneously on the same computer caused lag. Consequently, we employed two separate computers, one for audio and the other for visuals, both connected to the iPad via OSC, albeit on different ports

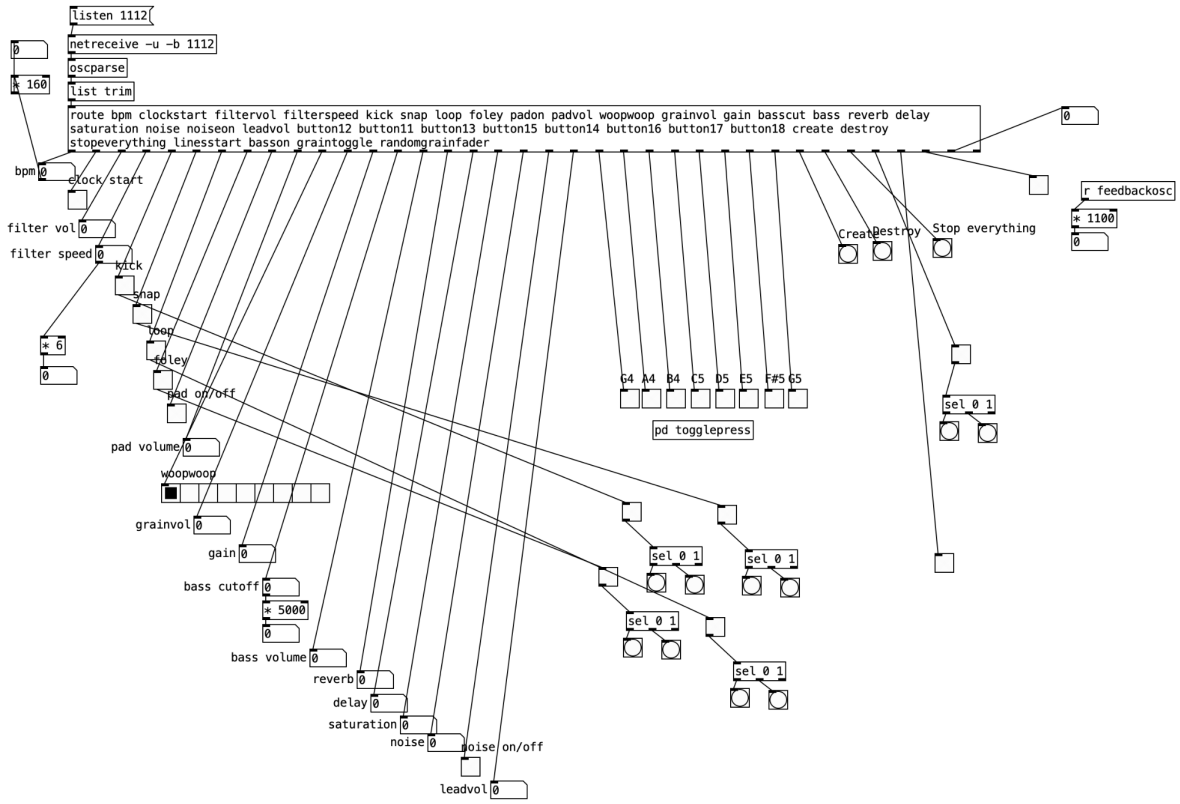

Figure 17: OSC controls.