**Final Presentation**

Abhishek Choubey

Andrea Sofia Vallejo Budziszewski

Computational Music Creativity

March 2023

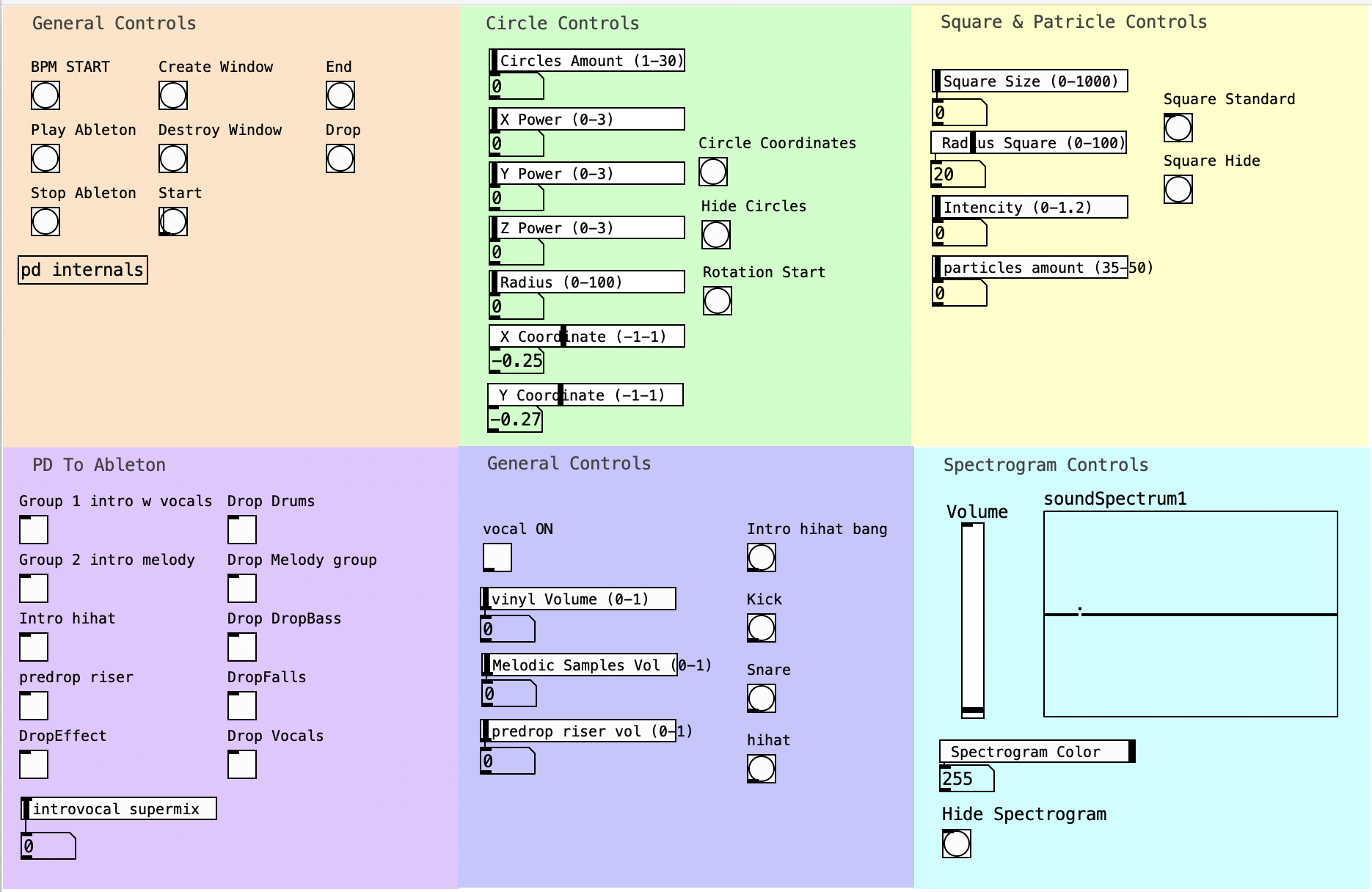


Figure 1. General view of the Pure Data (PD) patch.

**1. AUDIO**

**2. VISUALS**

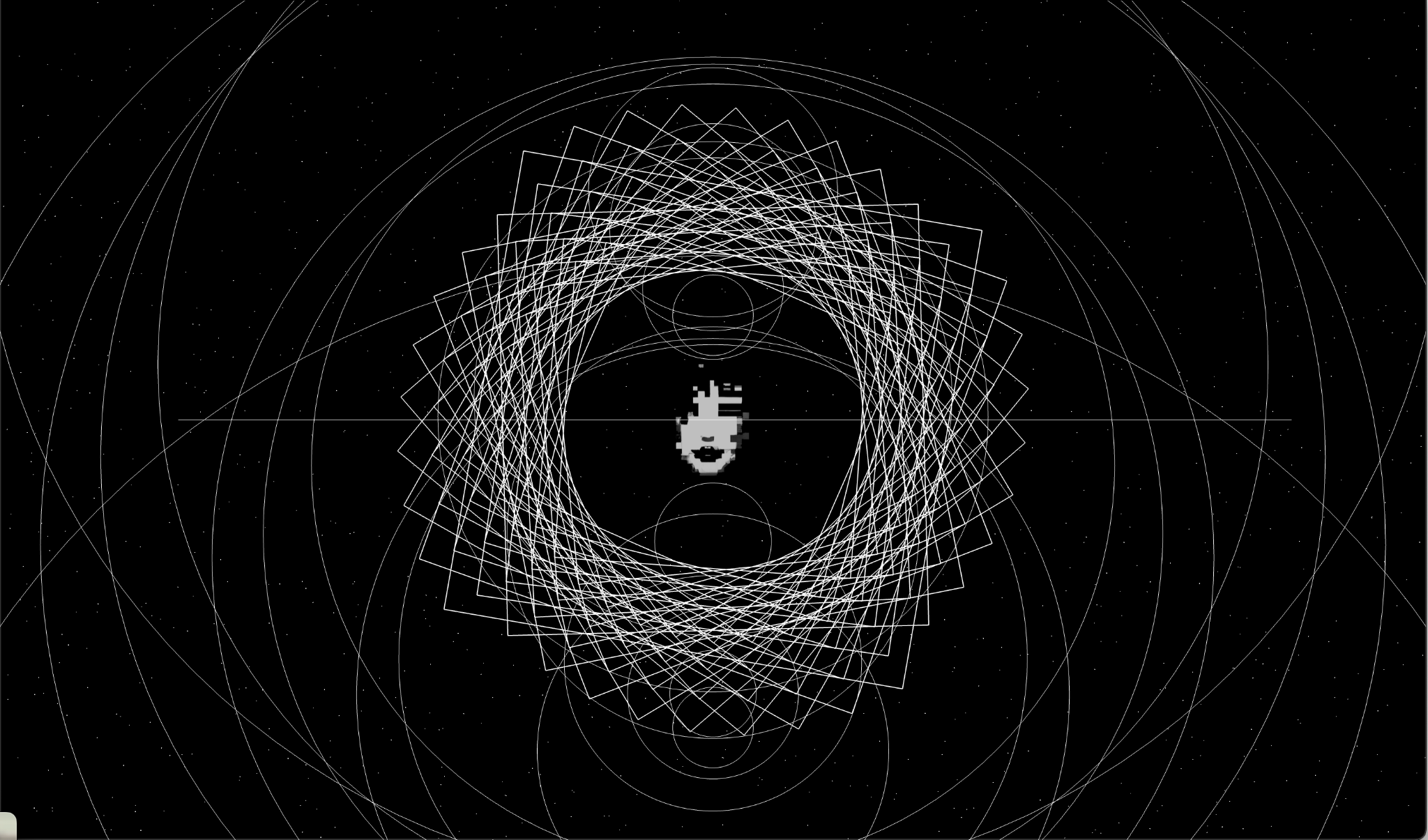


Figure 2.1. General view of the visuals.

The visual module is comprised of five individual modules.

**2.1. Image:**

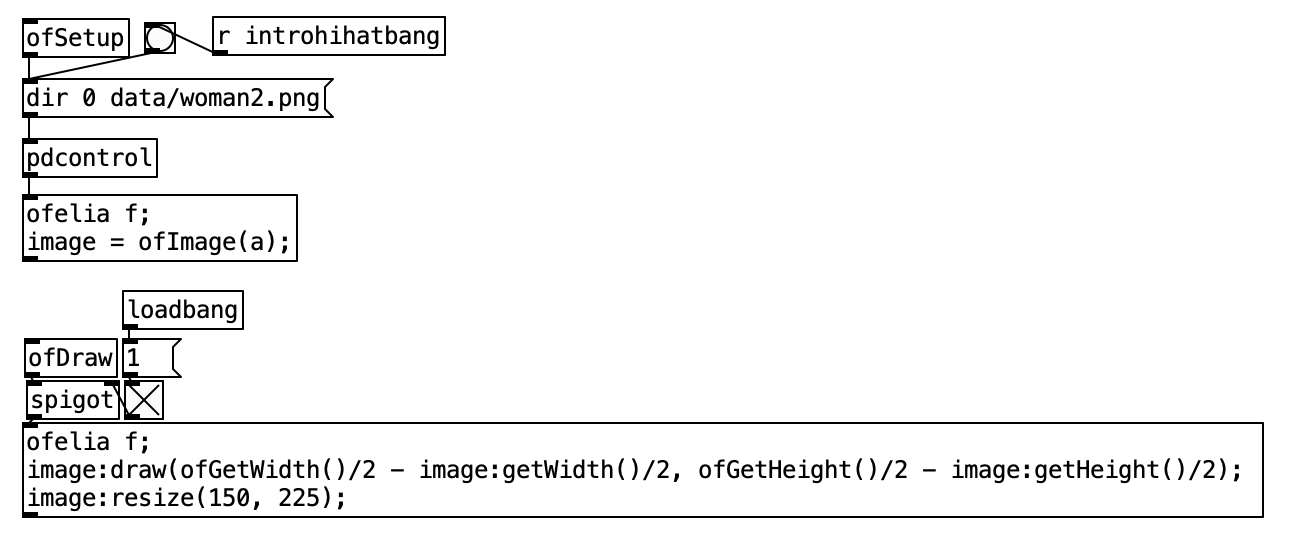


Figure 2.1.Image patch..

This part of the parch, opens an image (in the data folder, called woman2.png.) The bang makes the image reload which we used to make some kind of glitch effect.

**2.2. Circles:**

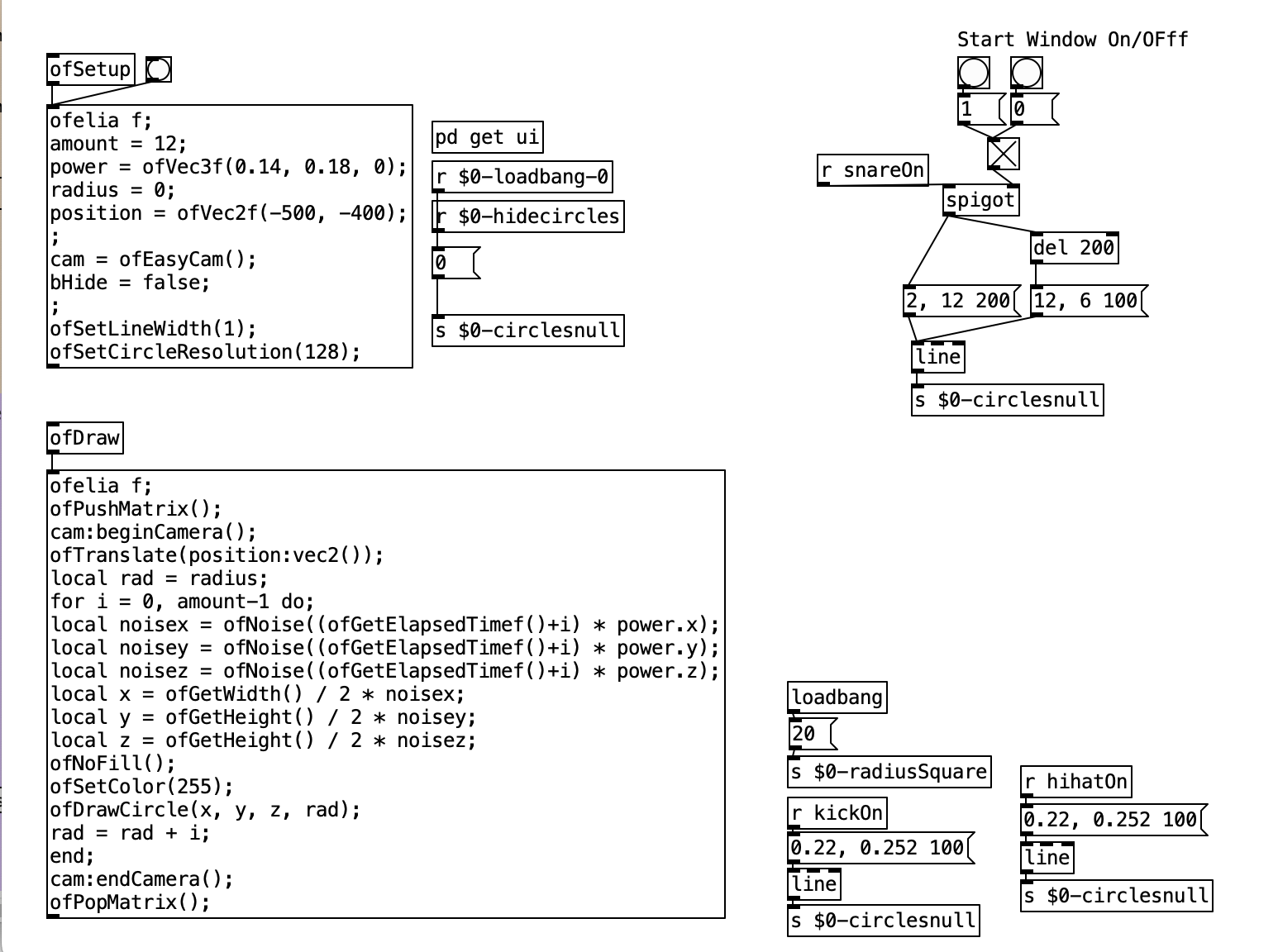


Figure 2.2.Circles main patch.

This code creates a 3D animation of noise-based circles that move and change in size over time. The circles are drawn using the ofDrawCircle function and the camera perspective is adjusted to provide a 3D view.

The variables defined are:

**amount**: A numeric value of 12, which appears to be used as a parameter in the for-loop.

**power**: A vector of three floating-point values (0.14, 0.18, 0), which are used in generating noise values later in the code.

**radius**: A numeric value of 0, which is used as the initial value for the variable rad in the for-loop.

**position**: A vector of two floating-point values (-500, -400), which is used to translate the origin of the coordinate system.

Other than these variables, the code initializes the camera and sets some graphical properties such as line width and circle resolution. The core of the code is a for-loop that iterates amount times. In each iteration, the code generates three noise values using the ofNoise function, which takes the current time (ofGetElapsedTimef()) and the index i as parameters multiplied by the power vector. These noise values are then used to calculate the x, y, and z coordinates of a circle to be drawn. The circle is drawn using the ofDrawCircle function with the calculated coordinates and the current value of rad. Finally, rad is incremented by i. The code also uses a ofPushMatrix and ofPopMatrix function to save and restore the state of the coordinate system before and after the for-loop. The cam:beginCamera() and cam:endCamera() functions are used to initialize and finalize the camera view, which appears to be an "easy" camera type.

**2.3. Squares**

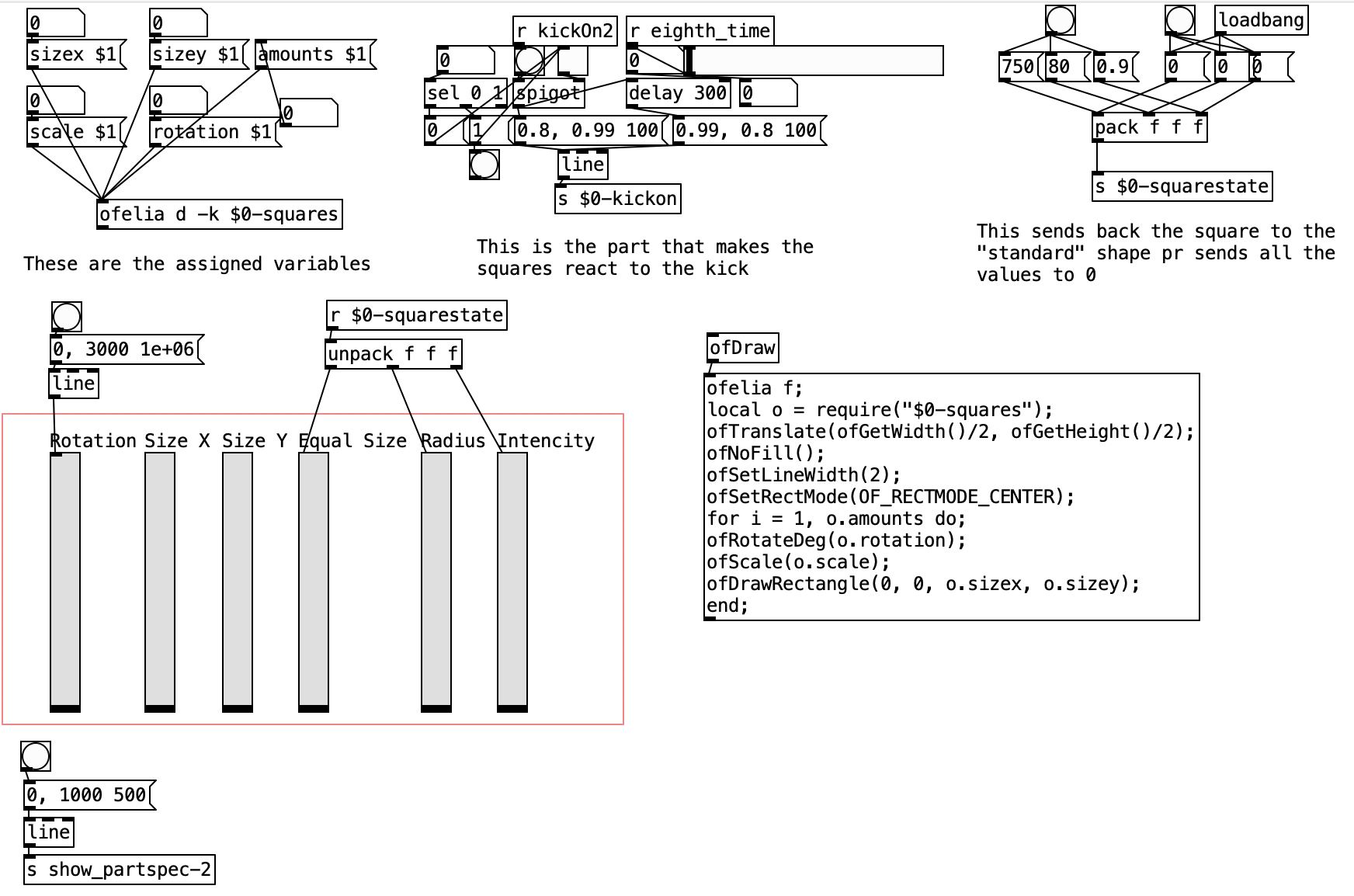


Figure 2.3.Squares main patch.

This code draws multiple squares with different sizes, rotations, and scaling factors using openFrameworks library in Lua programming language. The specific parameters for each square are defined in a module named $0-squares.

The variables defined are:

**o**: A local variable that imports a module or library that contains the following properties.

**amounts**: An integer value that determines the number of rectangles to be drawn.

**rotation**: A floating-point value that determines the rotation angle for each rectangle.

**scale**: A vector of two floating-point values that determine the scale of each rectangle in the x and y directions.

**sizex and sizey:** Floating-point values that determine the size of each rectangle in the x and y directions.

The code then translates the origin of the coordinate system to the center of the screen using the ofTranslate function, sets some graphical properties such as line width and rectangle mode, and initializes a for-loop that iterates o.amounts times.

In each iteration of the loop, the code rotates the coordinate system by o.rotation degrees using the ofRotateDeg function. It also scales the coordinate system by the o.scale vector using the ofScale function.

Finally, the code draws a rectangle centered at the origin of the coordinate system using the ofDrawRectangle function with dimensions o.sizex and o.sizey. Overall, this code generates a static display of a set of rectangles with varying rotation, scale, and size properties, all centered at the middle of the screen. The specific values for these properties are determined by the o variable, which appears to be defined in an external module or library.

**2.4. Spectrogram**

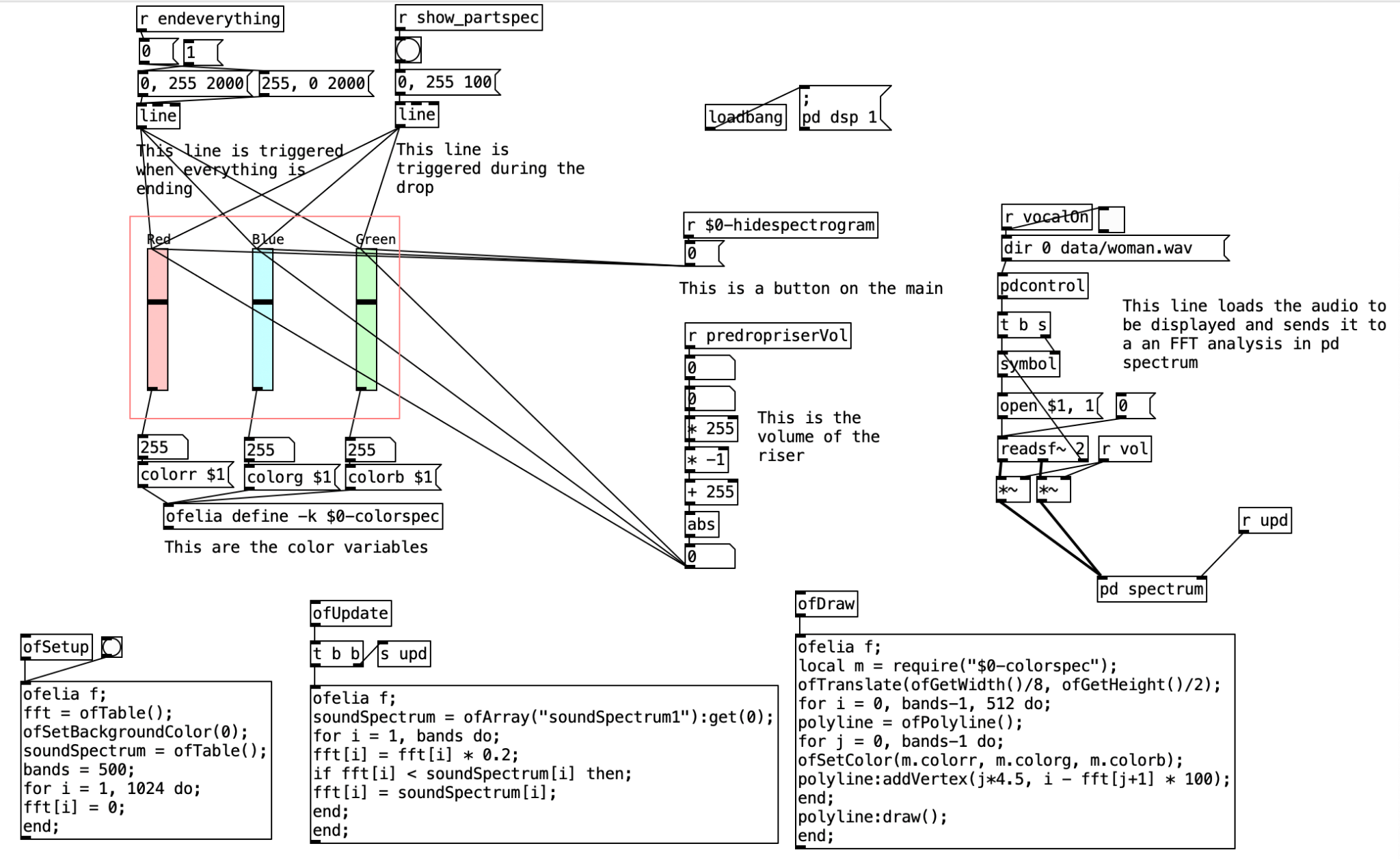


Figure 2.4. Spectrogram main patch.

This code visualizes sound spectrum data using polylines with color-coded vertices. The color scheme for the vertices is defined in a module named $0-colorspec.

The ofSetup function defines several variables and initializes an array of values to 0.

The variables defined are:

**fft**: An array of 1024 values that will store the frequency domain data of the incoming sound signal.

soundSpectrum: An empty array that will later store the amplitude spectrum of the sound signal.

**bands**: An integer value that determines the number of frequency bands in the sound spectrum.

The function sets the background color of the display to black using ofSetBackgroundColor and initializes all values in the fft array to 0 using a for-loop.

The ofUpdate function updates the fft array based on the incoming sound signal. The function starts by retrieving the amplitude spectrum data from an external array using the ofArray function. It then scales each value in the fft array by 0.2 using the expression fft[i] = fft[i] \* 0.2. Finally, it compares each value in the fft array to the corresponding value in the soundSpectrum array and sets the fft value to the soundSpectrum value if the soundSpectrum value is greater.

The ofDraw function generates the visual output based on the fft and soundSpectrum arrays. The function starts by importing an external module m that appears to define color properties. It then translates the origin of the coordinate system to the left edge of the screen and the middle of the screen using ofTranslate.

The function initializes a for-loop that iterates over frequency bands in the sound spectrum with a step size of 512. In each iteration, it generates a polyline using the ofPolyline function and adds a vertex to the polyline for each frequency band in the sound spectrum. The x-coordinate of each vertex is determined by the frequency band index multiplied by 4.5, while the y-coordinate is determined by subtracting the corresponding value in the fft array from i and multiplying the result by 100. The polyline is then drawn using the draw method.

Overall, this code generates a dynamic display of a sound spectrum that updates in real time. The visual output is a set of vertical lines that vary in height based on the amplitude of the sound signal at different frequency bands. The color of the lines is determined by an external module that defines RGB color values.

**2.5. Particles**

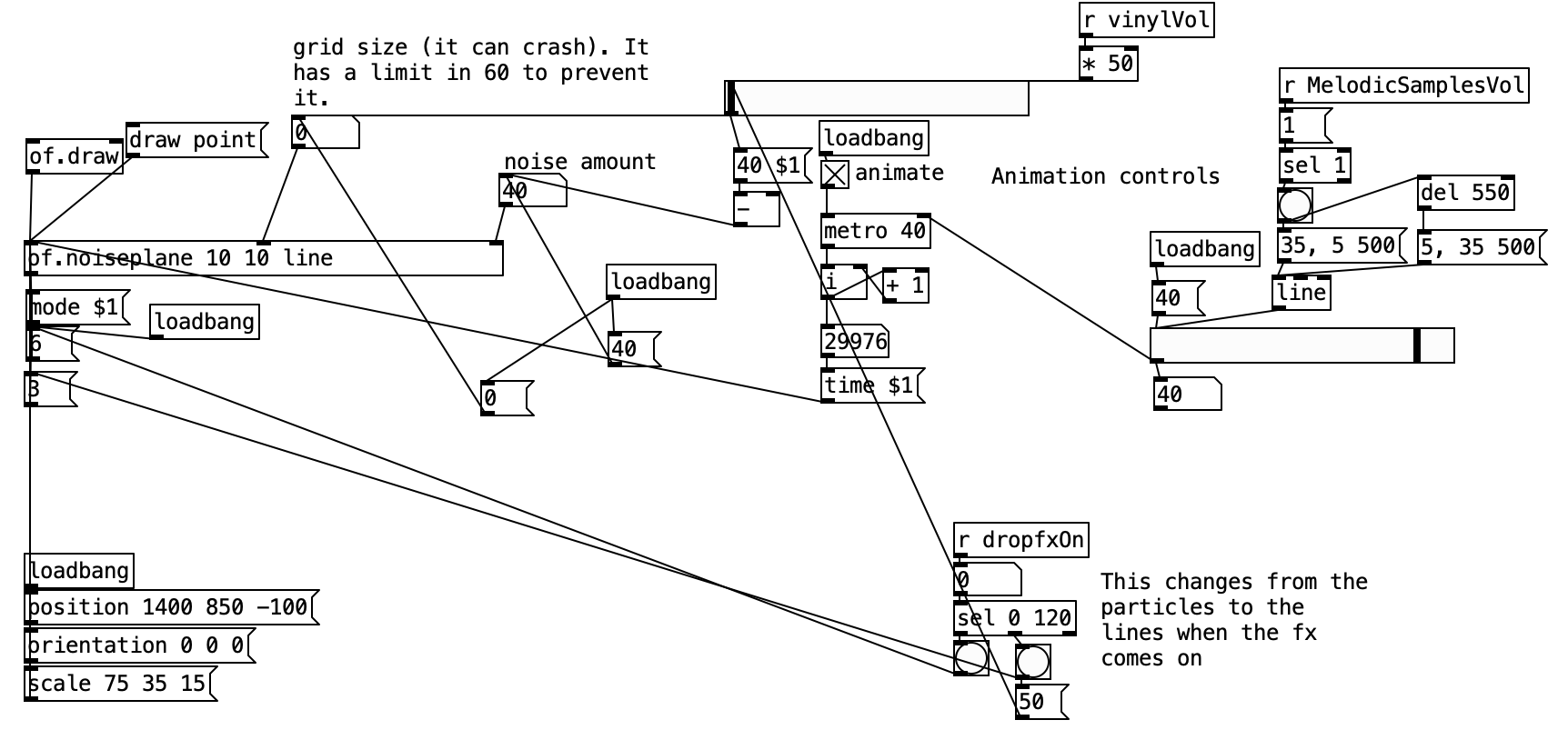


Figure 2.1.Patricles main patch.

The of.noiseplane defines a Lua class called "M" that generates a 3D plane mesh with noise-based animation. The mesh can be manipulated in terms of position, orientation, scale, and texture mapping. The mesh can also be rendered in different modes such as fill, point, and line. The code can be controlled through various functions and arguments.

**3. OSC / MIDI MESSAGES**

We established a connection between Ableton and Pure Data by utilizing OSC and MIDI Messages. Specifically, Ableton transmitted MIDI messages to Pure Data (as seen in Figure 3.1), while Pure Data reciprocated by sending MIDI messages to Ableton (as seen in Figure 3.2).

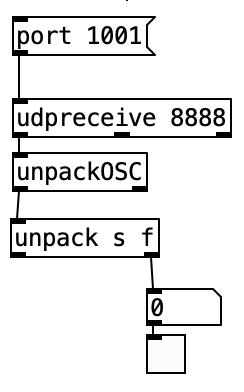
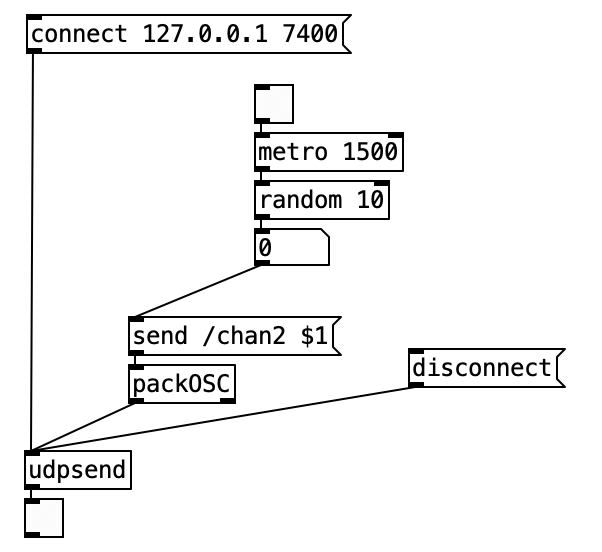
 

Figure 3.1 .MIDI Messages patch .Figure 3.2.OSC messages patch.