David Borts, Anna Lucas
Advised by: Professor Felix Heide, Vivien Nguyen

Modular Video Synthesizer

Abstract

The Modular Video Synthesizer is a virtual instrument for interactive video synthesis, drawing its aesthetic and design philosophy from the Eurorack format for audio synthesis. The instrument consists of 12 panels, made up of different 5 panel types, that users can control and wire together to generate rich and dynamic video in real time. Just like their physical counterparts, no two modules are linked together by default; the path that data takes from input to output is entirely set by the user.

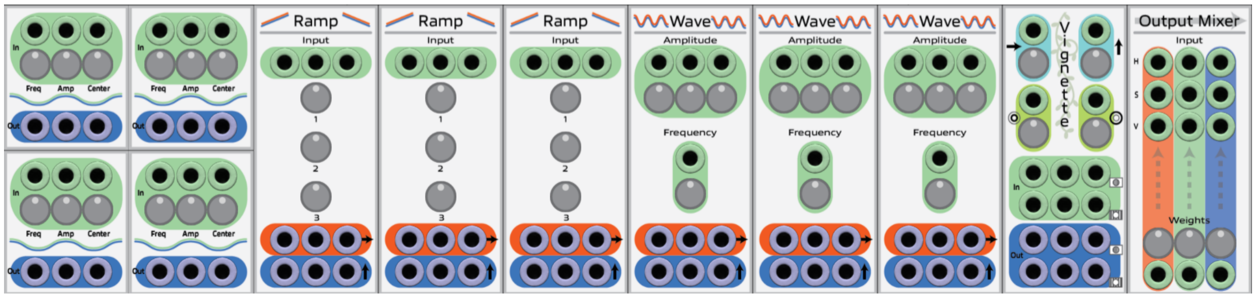
Introduction

Since their invention in the mid-Twentieth century, modular synthesizers have enjoyed a cult following of artists, hobbyists, and engineers. Through the repeated accumulation of many simple signals and mathematical operations, these instruments offer the immense power and flexibility to produce almost any combination of frequencies imaginable. For many, the downside of a daunting interface and steep learning curve were well worth the unrivaled flexibility, dwarfing the creative potential of more simple mass-market instruments that heavily constrained user behavior. In other words, modular synthesizers offer ***control***. No connections are hardwired; no tools are obfuscated from the user under the hood. The instrument becomes whatever it is wired to be.

However, modular synths are notoriously inaccessible in price, not just in utility. Today, a single module can retail anywhere from $200 for simpler designs to over $500 for more advanced varieties. Enter virtual modular synthesizers. In recent years, a number of digital emulations of classic analog modular synthesizers have entered the market, retailing at a fraction of the cost. Notable examples among these include VCV Rack, which boasts a library of thousands of audio synthesis modules and robust SDK that allows anybody to contribute, or Lumen, which is an extremely polished emulation of an analog video synthesizer. These have opened up the world of modular synthesis to more people than ever before, and are the inspiration for this project.

This project attempts to embrace this spirit of open-source collaboration that has been growing in the modular synthesis community. We have designed a virtual emulation of an analog modular video synthesizer that anybody can use to get familiar with the paradigm. To simulate real modules arranged in a case, an interactive GUI was created fully from scratch using THREE.js, including turnable knobs, patch cables, and custom-made sprites.

Features



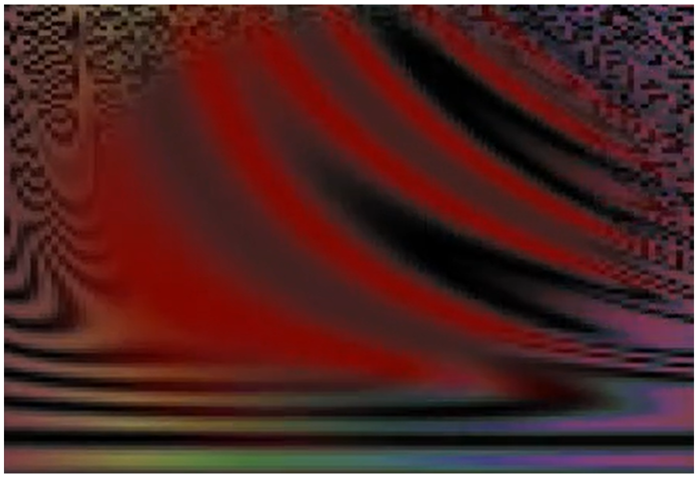
The foundation of the Modular Video Synthesizer are its two generative units: the Ramp and the Wave (pictured above). The former is the simplest of the two. It takes a single input, corresponding to its maximum value, interpolating between 0 and this value, depending either on a given pixel's vertical or horizontal position. The ramp proves immensely convenient for fading between colors or any other possible parameter. The Wave module follows a similar structure, taking frequency and amplitude inputs, which it uses to parametrize its oscillation in either the x or y direction.

Ramp

Wave

Beyond the ramp and wave, the LFO is the most indispensable module in the synthesizer, as it is the sole source of all dynamic modulation in the instrument. The LFO has the simplest inner workings of all, yet can produce some of the most complicated effects. The module output varies periodically over time, irrespective of any individual pixel's position on the screen. This oscillation is capped off at 20Hz - well below the audio frequency range - and can also be adjusted by frequency and amplitude inputs.

Implementation



Example output

Our screen object is fundamentally a sprite that refreshes with a new texture every frame. The source of the texture is a flattened array of 8-bit unsigned integer rgba values that we update and then pipe into a THREE.js DataTexture object. The source of a color at a pixel is obtained by simulating the path of the signal through the linked modules, similar to a directed acyclic graph with the output module as a singular root. Each output port stores a copy of the signal, scaled between 0 and 1, that exists from doing that module's transformations on its input. Each module's data must be invalidated upon a change in timestep, and if the module is needed during recursive traversal and has not yet been updated during that time-step, updated values will be calculated. The nature of the module calculations must be recursive because we require fully updated data from all previously connected modules "before" updated data can be produced at the output ports of a module. Unfortunately, this means that the inadvertent creation of a directed cycle with wires will cause an unresolvable dependency error; elementary directed-cycle checking will be implemented before the presentation.

In each module, the data transformation is relatively simple, and consists of something like setting the output to the sin of one of the coordinates scaled by one of the inputs. Optimization is generally not implemented due to the fact that the pixels tend to evolve in such complicated ways in space and time that optimizations for most of the modules are difficult or infeasible without changing fundamental data structure.

Linkage is handled directly from port to port. Since it may be a feature of our final product to have multiple inputs able to wire to a single output, the wire sprites are owned strictly by the input ports, and the only meaningful use of the output ports' link pointer is during wire creation and deletion. In the back-end these are represented by instances of the Button class, and Button inputs are sometimes supplemented by Knobs. Both Knobs and Buttons are children of the Module class, so their positions are relative to the parent module. While module spacing is mostly hardcoded, knob and port spacing are largely automatic and operate by parsing a set of specially formatted arrays in the Module_Map class. Part of the benefit of this is easy blocks of the same component, and convenient ways to insert port-size or knob-size gaps into the design grid (represented by zeros in the grid elements of the map, as opposed to ones) as well as change the configuration of rows vs columns and auto-center rows with irregular numbers of elements.

Implementation Challenges

Sizing and scaling elements of the GUI properly were some of the most time-consuming challenges of this project. All scaling and positioning units are now successfully in screen-percent instead of pixels, and most are controlled by constants near the top of a js file instead of manually hunting through the code to change "magic numbers". However, some changes may need to be made to the settings of how THREE.js is scaling the sprite textures, as we have noticed the shorter wires tend to have a lot of blur and edge artifacts when they depart too far from the original wire texture's aspect ratio. We suspect we will need to change some of the built-in settings for the sprites in order to preserve the clarity of most of the original PNGs used for textures. It is also true that the wire scaling and positioning code is overly complicated, and it tended to sink a lot of time in offsets that varied with every edit of the image. We also found that the angleTo Vector3 function only returns positive values regardless of which order the vectors are passed in, so the special case of an upwardly rotated wire needed to be handled explicitly.

The custom knobs were mainly challenging to keep the rotation and inner data value in sync to allow live updating while the knob was turning, while also clamping them both to a specific range. The scaling of the data propagating through the backend was also challenging in general, as we wanted default / "middle" knob values to correspond to settings that were already in a range that was useful to an inexperienced user, while maintaining clamp limits on the effects that made sense given what the effects were. For example, in Vignette, the minimum and maximum radii selectable are those close to the center and close to the outer corner. Yet these must correspond to knob-data or input between 0 and 1, which then correspond to a rotation between -2 and 2 radians, and the inner radius must not be able to grow larger than the outer regardless of what parameters the user has set the knobs to. We must also save both the knob-data and the y-value of the click in order to properly interpolate during up and down drag events - the knob turn is relative to the original rotation of the knob plus a scaled version of the distance between the original click Y and the current Y.

Reflections

Out of our original goals, many were achieved at higher speed, clarity and responsiveness than originally predicted. Specifically, we are very proud of the program’s ability to live-update

and produce extremely fast, responsive, complex graphics. It was a big concern whether this real-time video generation would be possible given the large amount of minor computations that would need to be performed, but our implementation is very fast and robust.
Some of the modules we expected to implement we chose not to for the sake of time and because after we started experimenting with the program we realized they wouldn't be as useful as certain others. One module we struck from our MVP was a noise generator; another was a particle generator (which would have worked by generating shapes, say, circles, and adjusting their size, quantity, position and other attributes with the oscillator). While these can be useful, we found the ability to combine horizontal and vertical versions of effects was much more pressing and rewarding in potential.

However, we were able to move far past an MVP by implementing three types of filter (Ramp, Wave, Vignette) with several tunable parameters plus a signal-combining output mixer. We are also very happy with how far our graphics and GUI have evolved past the bare-bones gray boxes we were using during feature testing. Specifically, the ease of use of the current program is far beyond what we were attempting to use during testing - all of the ports are well-spaced, clearly labeled, and easy to wire together.

User Feedback

One piece of feedback that stood out to us was when our friend mentioned that the Wave filter did not have an option to move up and down the page. We could incorporate a phase parameter or offset into the sine calculation that would allow people to achieve a vertical-scroll or horizontal-scroll effect with the wave positioning on the screen. Currently, our waves are locked such that even when the wave period changes, x=0 will always correspond to 0 amplitude, and the waves will "squish" toward the bottom or left side.

Another user opinion that was interesting to us was the idea that despite hearing about the concept and how the filters could be put together, our friend had no idea just how spectacular or complex the produced visuals really were until seeing them. This shows us that we need a much better system for tutorial/introduction to really be able to "sell" the concept to interest a potential user, and/or a default configuration of wires that load in rather than an empty board.

Next steps

One of the first things we do after submitting will be to improve our detection and handling of directed cycles among the wires. Another will be a tutorial made of javascript popups, and some changes in the available modules. We had really wanted to implement a Gaussian blur module and some of the other types of filters from Assignment 1 to complement Vignette. There is some miscellaneous polishing to do in the screen panel layout, and a possibility of adding a panel of switches for global toggle effects.

Contributions

While we worked collaboratively for large parts of the main ideas and design, there were parts of the work for which one of us tended to specialize.

David provided useful background knowledge of modular audio synthesis and the design of the existing physical components. He implemented a lot of the back-end math, data updating, linked module traversal, and frame update code. With David's skills in Photoshop and Illustrator, virtually all of the sprites are textured with original art. He also handled the usage of npm and the hosting of the github demo.

Anna added robust functionality for the GUI: link/unlink to ports, wire sizing, spacing, and colors; custom knobs through handling of base mouse events; procedurally spacing GUI elements with module maps relative to screen size; and designing the DataTexture screen refresh and interfacing. She also contributed significantly to the back-end recursive design and array handling.

Citations

- Port image (Open Source)
  - https://www.reddit.com/r/whatisthisthing/comments/6xezgw/what_do_you_plug_in_the_blue_input/
- Three JS documentation
  - https://threejs.org/docs/
- Modular Video Synth tutorials
  - Video Synth Techniques 1: Generating Horizontal & Vertical Waveforms
    - https://www.youtube.com/watch?v=sgnpVI9qPVo&ab_channel=LZXIndustries
  - Video Synth Techniques 2: Creating Shapes & Patterns
    - https://www.youtube.com/watch?v=EtJ-sCbEZOs&ab_channel=LZXIndustries
- RGB to HSV Conversion Code
  - https://stackoverflow.com/a/31851617
- Wire Color List
  - https://colorswall.com/palette/102
- Cakery Bakery - 2D THREE.js GUI implementation
  - https://github.com/cz10/thecakerybakery
- Feedback solicited
  - Thanks to Erik Bahnson for feedback provided on the project.
- Course Documentation
  - The starter code was very helpful in terms of publishing to Github Pages.
- Click Event Misc Help
  - https://javascript.info/mouse-drag-and-drop
  - https://stackoverflow.com/questions/7956442/detect-clicked-object-in-three-js
- Knob Inspiration & Help
  - https://www.cssscript.com/touch-enabled-knob-input-javascript-knob-input/
  - https://codepen.io/jhnsnc/pen/KXYayG
- Favicon (Beyond MVP, but we will most likely use this)
  - https://favicon.io/