# Generating Sounds That Don't Exist

Utilizing a modified StyleGAN3 to generate reverse-transformable spectrograms

Jon Henshaw
Computing and Software Systems
University of Washington, Bothell
Bothell, WA, USA
jhenshaw87@gmail.com

## ABSTRACT

We design a lossless spectrogram format and train an instance of StyleGAN3 on a set of these images in an attempt to provide proof-of-concept for the backbone of a generative musical sampling synthesizer.

## CCS CONCEPTS

• **Computer systems organization → Neural networks**; • **Applied computing → Sound and music computing**.

## KEYWORDS

NVIDIA, StyleGAN3, Generative Adversarial Networks, GAN, Artificial Intelligence, AI, Machine Learning, ML, Machine Intelligence, MI, audio, music, sound, synthesizers, sampler, spectrograms, Fast Fourier transform, FFT, CMYK, mid/side processing, stereo

**ACM Reference Format:**
Jon Henshaw. 2022. Generating Sounds That Don't Exist: Utilizing a modified StyleGAN3 to generate reverse-transformable spectrograms. In *Proceedings of CSS 486 - Machine Intelligence (CSS 486).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Music is one of the most prevalent fields of exploration at the cutting edge of technology. At every technological leap in human history, you will find people utilizing those ideas to create new instruments and new types of musical sounds. Particularly in the past few decades, music technology has become extremely prevalent, cheap, and easy to use. One market that has arisen in music technology is that of 'presets:' Professional sound designers spend some time developing settings for a synthesizer, then sell those settings to music producers and songwriters who want easy-access to great sounds. But what if we could create a synthesizer that generated a functionally infinite number of its own presets? Many synthesizers come with a 'randomize' button—but applying random values to each setting on a synthesizer is not guaranteed to make a pleasant or even interesting sound, especially compared to curated patches designed by a professional sound designer—and it may be very difficult to find a sound that is appropriate for your present needs: if you're looking for a snare sound, how likely is it that you will get one by hitting the 'random' button on a synthesizer? The chance is astronomically low. The idea to use machine learning for this purpose arose out of a need to solve these types of problems: How can we develop near-infinite, unique, professional, and customizable sounds?

"The style-based GAN architecture yields state-of-the-art results in data-driven, unconditional, generative image modeling."[9] You may have seen the website thispersondoesnotexist.com, where you can reload the page and a different, randomly-generated face will appear each time. None of these pictures are pictures of real people, but are fabricated by blending 'styles' of different pictures together. However, StyleGAN3 can be trained on any type of image set—not just one of human faces. If trained on spectrograms, this architecture could possibly generate its own spectrograms—and if these spectrograms had reverse-transform capability, those generated spectrograms could then be transformed into audio. Thus, while StyleGAN3 takes in a fully randomized input, it could be trained on a set of 'good-sounding' spectrograms, in order to map those random input coordinates to some point in 'good-sounding'-space—unlike what happens when you simply hit the 'randomize' button on a synthesizer.

StyleGAN3 also allows for directed blending. We can take any randomized image, and 'move it in the direction' of some characteristic we desire. For example, if we look at the set of input vectors that produce an output of a snare sound, we could find the 'snare' direction of our input space. Then, after generating any kind of sound, we could blend the 'snare' style into it, creating an extremely unique snare sound. The same could apply for any type of instrument, envelope, filter, effect, or even musical genre! One could even find the 'direction' of each note on the keyboard and generate a different sample for each one, instead of timestretching a single sample across multiple keys. If this type of engine were used as the backbone of a sampling synthesizer, it would be able to generate a near-infinite number of professional-sounding, unique, and user-customizable samples for itself.

We do not attempt to build such a synthesizer in this paper. However, we do some preliminary proof-of-concept work to determine whether such a project might be feasible by designing a custom, high-fidelity, reversible spectrogram, with the intent of training StyleGAN3 on a set of these spectrograms to see if it can generate usable synth sounds.


*CSS 486, Fall, 2021, University of Washington, Bothell*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 2 THE SPECTROGRAM

A spectrogram is defined by the OED as "A photographic or other visual or electronic representation of a spectrum."[11] On Wikipedia, it is defined similarly as "A visual representation of the spectrum of frequencies of a signal as it varies with time."[5] Thus, it seems there is no specific, technical definition of a spectrogram, and as such, many spectrograms vary wildly in implementation. However, there are some similarities shared: Typically, frequencies are represented on the vertical axis, with time on the horizontal axis. Then, windowed FFTs are performed on the signal—to decompose it into its constituent frequencies—and higher magnitudes of the coefficient for any frequency corresponds to a brighter pixel value in the image. Some interpolation or weighted multi-windowing is often used to create smooth transitions from one column of the image to the next. However, there are three general problems that most spectrograms exhibit in their representation of the frequency spectra of a given signal, which must be addressed if we are to succeed.

The first is bit-depth: most professional-quality audio applications produce sounds represented by floating point values on [-1,1] at a bit depth of 24, 32 or 64. In a .png or similar image file, values for each color channel per pixel are 8-bit integers on [0,255]. Further, just because an audio signal is on [-1,1] does not mean that its FFT coefficients will be as well, meaning that a much larger range may need to be scaled down to [0,255]—and with finite-precision, this may lead to loss of accuracy. With both of these issues, we may encounter a huge drop in fidelity.

Second is loss of sign, phase, and stereo information: most spectrograms simply use the real part of the FFT, discarding the phase information provided by the imaginary part; further, only magnitudes of the frequency coefficients are considered, which loses us even more phase information—and the signal is typically folded down to mono before analysis, as we would require two images to be able to fully describe a stereo sound with this schema.

Thirdly, as a result of this loss of data (and any potential interpolation or weighted multi-windowing), most spectrograms are irreversible: the original audio generally cannot be constructed from its spectrogram, as so much of the original information is lost. This presents a real problem to the conceit of our project, and must be addressed with the design of a custom spectrogram format.

### 2.1 Designing a lossless Spectrogram

The most pressing change we must make is substituting float32s for the int8s typically saved in an image file. So not only must we create a custom spectrogram, we must create a custom image format. This means we will have to modify StyleGAN3 to take in a different data type than it is presently designed for. Increasing the data type size in this way will also surely make training our algorithm more time-consuming. However, it will be necessary if we are to generate anything worth listening to outside of a chiptune context.

The second change we must make needs to recover both phase and stereo information. We could consider the left and right audio channels of a stereo file, and track magnitude and phase of all frequencies for each respective channel in the form of the real (cosine) and imaginary (sine) parts of that channel's FFT. As an audio signal is inherently real, the negative frequencies will always be equal to the positive frequencies, and we will be able to save some processing time, storage space, and room in the image ($\frac{n}{2} + 1$ samples are returned by RFFT as opposed to the $n$ returned by a regular FFT), by running scipy's rfft()[4]. And while the following may not be the case for each individual window due to truncation: most professional audio contains a DC offset of 0, so we can discard the zero-frequency information as well, assuming it to always be 0.

But should we be looking at the left and right channels separately? In music production, it is often more useful to think about the 'middle (M)' and 'side (S)' channels of an audio signal, as opposed to the 'left (L)' and 'right (R)' channels. This is because the mid and side channels can be used to describe the stereo width of a sound: a feature of the sonic palette that makes up that sound. These four simple equations give a powerful insight into the *relationship* between the left and right channels of a stereo file, instead of what they are each doing on their own:

$$M = (L + R)/2 \tag{1}$$

$$S = (L - R)/2 \tag{2}$$

$$M + S = (L + R)/2 + (L - R)/2 = \frac{L}{2} + \frac{R}{2} + \frac{L}{2} - \frac{R}{2} = L \tag{3}$$

$$M - S = (L + R)/2 - (L - R)/2 = \frac{L}{2} + \frac{R}{2} - \frac{L}{2} + \frac{R}{2} = R \tag{4}$$

Because of how these values are computed, the 'mid' channel is often referred to as the 'sum,' while the 'side' channel is referred to as the 'difference.' What these represent is what you get when you fold a stereo sound down to mono (mid), and the information lost when you do so (side). In this way, mid/side processing can affect the stereo image of a sound. So, instead of using the L and R channels in our spectrogram, we will use the M and S channels, which may give StyleGAN3 a more descriptive insight into the quality of any given sound.

However, this does not reduce the dimensionality of our data. If we were to use the L and R channels, we would need to store both magnitude and phase information for the frequencies comprising each channel in the form of the real and imaginary parts of the RFFT. The same is true, even though we are instead looking at the M and S channels. This means we will still need a total of 4 color channels to store all the information required for a reverse transform: $M_r,\ M_i,\ S_r,\ S_i$. This leads us to using a 4-channel image.

There are a few different standardized 4-channel image formats. One is RGBa, and another is CMYK. It makes more sense to think of using CMYK, as RGBa does not treat each channel equivalently (the 'a' channel is for transparency). Further, StyleGAN3 is only designed to accept 3- or 1-channel images (RGB and BW), so we would not be able to take advantage of some built-in functionality by choosing RGBa.

We then have that our image format will wind up being a 3D array of float32s, the first dimension being the height of the image, the second being the width, and the third being the four color components of that pixel in 'CMYK-float' color space. We will output these using numpy's save() and load() methods[3], without pickling, to maintain version flexibility.

## 2.2 Conversion to .png

However, it now seems that this conceptual image format we've developed for our spectrogram cannot actually be 'seen' by human eyes. Because of this, we have decided to separately develop a conversion from our conceptual image format to .png. This will be useful for determining whether our spectrogram produces images of a recognizable qualia, and whether our trained instance of StyleGAN3 produces spectrograms that match said qualia. In this conversion, we discard the sign of all values, scale their range to [0,255], and (because .png does not support CMYK) perform CMYK—>RGB conversion[13]:

$$R = 255 * (1 - C) * (1 - K) \tag{5}$$

$$G = 255 * (1 - M) * (1 - K) \tag{6}$$

$$B = 255 * (1 - Y) * (1 - K) \tag{7}$$

We then invert the colors, so that we see some light colors on a black background, which is easier on the eyes. Finally, we convert the vertical axis to a log2 scale, which is more intuitive for musical information due to the octave effect. A portion of one such image is displayed to the right. The low-to-high filter sweep at the very start can be seen on the left, and so can the harmonics cascading up. Increased brightness corresponds to higher magnitude of frequency, while shifting color represents shifts in the stereo image and phase. While these are interesting to look at, it is important to remember that these .png representations of our spectrograms are for human eyes only—we do not intend to use these to train StyleGAN3, as they re-introduce the problem of data loss preventing reverse-transform. However, our goal is to give StyleGAN3 patterns of 'brightness' and 'color' in our conceptual image format in the same way that it does with 3-channel images.

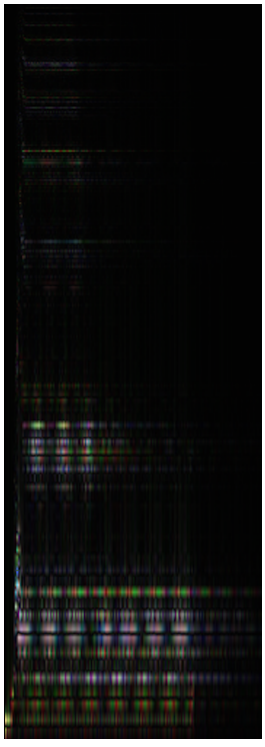


**Figure 1: Portion of a .png generated from a lossless spectrogram**

## 2.3 Audio File Similarity Function

Finally, we need some way to quantify how well our transform and reverse transform maintain fidelity of the original audio, or there will be no guarantee that our training set will contain valuable information, or that we will be able to reliably convert the generated spectrograms to audio. The algorithm for the similarity function is as such: Convert each audio file to mid/side. If they are of unequal lengths, pad the shorter one with zeroes. Find the cosine similarity between the two mid channels, and between the two side channels. Then, return the euclidean distance of the vector with those two values as its components, divided by $\sqrt{2}$ (so that the maximum similarity from this function is still 1).

$$\text{cosine\_similarity}(\bar{x}, \bar{y}) = \frac{\bar{x}^T \bar{y}}{||\bar{x}||_2 \cdot ||\bar{y}||_2} \tag{8}$$

There was initially a numerical issue with empty- or mono- audio files, as they will have entirely empty channels, but the cosine similarity between the zero vector any any other vector is undefined, due to the denominator being multiplied by zero. The first approach to fix this was to choose $\max(||\bar{x}||_2,\ \epsilon)$ for both vectors when calculating the denominator of the cosine similarity, where $\epsilon$ is equal to the machine epsilon of the floating point system we are using (the smallest value subtractable from 1 given that system's finite precision). This solved the divide-by-zero issue, but caused surprisingly low similarities ($\sim 0.65$) for mono files and their transformed/reverse-transformed counterparts. Since our reverse-transform always creates a stereo file, and there is some tiny error when going back and forth, mono files do not end up with a completely-empty side channel after the transform. So instead, we replace any *completely* empty channel by one filled entirely by $\epsilon$, and our mono files are now evaluated appropriately.

There is some variability in the similarity between original and reconstructed files, but when using a window size of 1024, all similarities have been found to be $\geq 0.90$, with the largest found so far being $\sim 0.99997$. There is almost no perceptible difference when listening back to the files besides some low-end distortion which would likely go unnoticed by an average listener (and is probably caused by our assumption that each individual window has a DC offset of zero). Contrapositively, we also find that similarities of completely different files are exceedingly small, returning values of an entirely different magnitude; when comparing a vocal sample with a drum loop, we find a similarity of $\sim 0.00416$. That means our transform—and our reverse-transform—work!

# 3 THE TRAINING SET

Now that we have a design for the lossless spectrogram, we must generate a training set to be used with StyleGAN3. During the generation of each spectrogram in our training set, we re-sample any audio file with a sample rate other than 44.1kHz (industry standard for digital audio) to 44.1kHz, using numpy's interp()[2] function. We do this in order to present equivalent spectra across our entire spectrogram set. However, as we perform no filtering, this may result in some aliasing noise, or the prevalence of a band of the previous sampling rate's nyquist frequency. As explained further below, we will be generating our own training set, but this stipulation has been implemented to prevent loss of generality for any future research.

### 3.1 Required Characteristics of Data Objects

StyleGAN3 requires that every image in the training set be square, and have the same power-of-2 resolution. Because an RFFT window size of 1024 will allow us to capture frequencies down to 20Hz (the lower range of human hearing) while still retaining a good amount of time-domain information between columns of the image, this makes a reasonable choice for the window size for our purposes. As we then proceed toss out the 0-frequency information and store each window vertically in our spectrogram, this will give our lossless spectrograms a column height of $\frac{1024}{2} + 1 - 1 = 512$, which is a power of 2. $512 \times 512$ is not the largest image resolution supported by StyleGAN3 (that would be $1024 \times 1024$), but that is okay for this preliminary work, as our main goal is to discover whether the general idea is feasible, not to produce a finished product.

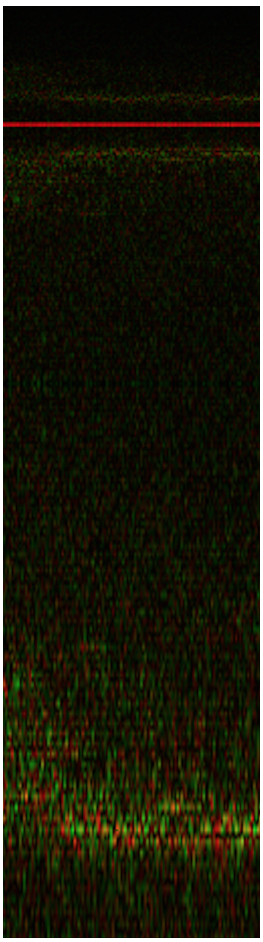


**Figure 2: The red band near the top of this spectrogram portion represents the nyquist frequency of its previous sampling rate of 11kHz**

With this resolution in mind, we can truncate each audio file (or pad it with zeroes) so that it will have a total length precisely equal to 512 windows of size 1024. This corresponds to each audio file in our training set containing a total of $1024 \times 512 = 524,288$ samples. At an audio sampling rate of 44.1kHz, that corresponds to $\frac{524288}{44100} \approx 11.8886$ seconds of audio per spectrogram. Almost 12 seconds seems like a reasonable length for a good bass synth sample, and even for many pad-type patches, though many of those can have automations that affect the character of the sound for multiple minutes. It will certainly be enough for a pluck sound, though one of these may not make very much happen in the spectrogram due to being too short and not taking up very much of the available space in the image. So where will we get our audio files from to construct our training set?

### 3.2 Generating Synth Samples

There are many free audio sample banks out there, and we have, over the years, obtained many of our own. However, as the goal is to eventually build the backbone for a synthesizer, we want to be able to use 'normal' synthesizer samples for our training set, instead of loops and kits, which is what constitutes the majority of most commercial sample banks. Thus, we decided to export 48 audio files for each 'pad,' 'bass,' and 'pluck' patch that comes standard with Reason Studios' Europa synth in Reason 11[15], which gave us $\sim 11,200$ audio files to work with. We sampled about half of each of these patches on every note in the even octaves (0 through 6), and the other half of the patches on the odd octaves (1 through 7). The samples were all generated using the maximum MIDI velocity of 127. The duration of each note is such that the tail of most of the synth patches will end before 11.8 seconds has passed. Because this is a tedious process, we did not sample a full range of velocities and note lengths, nor was care put into what pitch- or velocity- ranges might sound good on an individual patch. However, we do believe this will suffice for a proof-of-concept demonstration.

## 4 STYLEGAN

StyleGAN3 represents the present culmination of NVIDIA's work on improving the structure of Generative Adversarial Networks (GANs). The basic concept of a GAN is that it is an unsupervised machine learning model that learns to generate files similar, but not identical, to those in the training set. This involves simultaneously training two different neural networks: 1) The generator, and 2) The discriminator. These two networks are adversaries. The generator's job is to create images that look genuine, and the discriminator's job is to determine whether a given image is real, or a forgery.

During training, the discriminator is first fed images from the training set, and, using backpropagation and gradient descent, is taught to output a 1 for those inputs. Randomized noise is then fed into the generator, which outputs the best images it can from the knowledge it currently has. These images are fed into the the discriminator, which is then trained to output a 0 for those inputs. However, the backpropagation is also used to train the generator towards a direction that would have made the discriminator output a 1, so as to better fool it in later iterations. Eventually, the discriminator gets better at determining what makes a genuine image, but at the same time, the generator gets better at generating genuine-looking images. The result is that the discriminator will eventually converge to always output $\sim 0.5$, whether the inputted image is genuine or fake, because it knows what a real image looks like, but so does the generator, and the discriminator is unable to tell the difference. At this point, the discriminator can be discarded, and the generator can be used for its intended purpose.

The style-based GAN architecture[8] makes many modifications from a standard GAN. One of the main areas NVIDIA have focused on is disentangling the input space. Since the input space is completely randomized during training, GANs can suffer from poor image quality arising from dissimilar properties becoming abut within the input space. This can lead to the formation of features that are unrealistic, convoluted, or low-quality, when sampling at or around such a boundary. To better achieve disentanglement, the fully-randomized input is first transformed into an intermediate latent space, and then undergoes a series of affine transforms corresponding to the separate *styles* which control different layers of the generator's architecture. Each layer is called a *style block*, and

corresponds to features that typically arise during different image resolutions (tone, pose, expression, etc). Each layer also receives some uncorrelated noise to randomize ineffable qualities (individual hair placements, pores, etc.). NVIDIA also designs custom drivers to be compiled on the fly to better take advantage of their proprietary hardware.

In StyleGAN2[9], excess normalization had been found to create extreme maxima, resulting in droplet artifacts. This led to reconfiguration of the architecture to remove some of these normalisations. At the same time, they increased the size of the higher-resolution layers, as it seemed the highest-resolution layers were not affecting the style of the image as much as simply upsampling to a higher resolution. After this version, they also move away from tensorflow, instead utilizing pytorch for its GPU implementations, removing many of the compatibility restrictions arising from tensorflow.

With StyleGAN3[7], the signal processing methods have been revamped to better deal with aliasing arising from discrete edges like image boundaries and truncation by activation functions. This aliasing can lead to features becoming fixed early in the generative process, preventing subsequent layers from affecting the style of the image. Metrics have been completely revamped as well. Another change was to only round and scale the network's internal 'image' data to RGB at the very end of the neural network, so we will have fewer modifications to worry about with that respect.

Besides generating extremely realistic images, the layered style structure and improved signal processing methods are what make StyleGAN3 such an appropriate choice for our end goal. High-fidelity signal processing is of key importance for audio applications. And as we want to work towards building a synthesizer that is able to inject a variety of different styles into any given sound, the style-architecture is a natural fit.

## 4.1 Required Modifications

We have discussed what the main necessary modifications to StyleGAN3 for our project are. NVIDIA's license for StyleGAN3 allows for this type of modification. There were some easy parts, even if they took a while: Going through and removing all uses of PIL's[1] Image.open() or Image.save(), and replacing them with calls to our spectrogram methods; Replacing all data types of `uint8` or `float16` with `float32`; Searching for instances of the numbers `3` (used for validating inputs as 3-channel images), `255`, or `127` (for scaling or normalizing with respect to RGB), and removing or altering them as necessary.

StyleGAN3 has a dataset construction tool, which can be run on a folder of individual image files, and outputs a single, uncompressed .zip file for fast indexing during training. We modified this tool with our spectrogram methods, enabling it to accept a folder of .wav files. It then runs each .wav file through the transform to lossless spectrogram before exporting it to the uncompressed .zip archive. Our choice to output our spectrogram files without picking maintains synergy with this uncompressed fast-indexing strategy. The `gen_images.py` file was also modified to run the trained generator's lossless spectrogram outputs through both the .png transform and the reverse transform to .wav, saving each of these files to the output folder instead of the raw .npy, which is then discarded.

However, some of these modifications became much more difficult to implement than initially estimated, specifically updating the linear algebra methods in the image transform pipeline from using 3 channels to using 4. Most of these transforms involved simply adding an extra column and/or row to the transformation vectors and matrices: Scaling, instead of in 3 directions, in 4 directions. However, one of the transforms in this pipeline is a "hue rotation." This means that each pixel's 3-channel color is seen as being placed as a point 3D space, and is then 'rotated' around the unit Luma vector: $\begin{bmatrix}\frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}}\end{bmatrix}^T$. This presented a challenge to our understanding at the time, because as it turns out, a 4D object cannot be rotated around an axis, and necessitated delving into the subjects of exterior and geometric algebra.

## 4.2 4D Rotation

When rotating 2D object, all that is required is a single point. One can put their finger down on a piece of paper, holding that point fixed, and the paper can be spun in one of two directions about that point: clockwise (negative angle), and counter-clockwise (positive angle). However, in 3D, an object cannot be spun around a point by a single $\pm\theta$ angle, as there are an infinite number of directions to spin the object around a single point in 3D. This is why an axis is needed for rotating an object in 3D: The entire axis is held stationary instead of the single point. Similarly, in 4D, an object cannot rotate around a single axis alone. Instead, a 2-blade[16] must be used. A skew-symmetric 2-blade (which is required to construct a rotation matrix) can be calculated by the wedge product[10] of two column vectors, as defined by the difference of commuted outer products:

$$\bar{v} \wedge \bar{u} = \bar{v}\bar{u}^T - \bar{u}\bar{v}^T \tag{9}$$

The axis used for rotation in 3D defines what quantities are held fixed, and the 3D hue rotation was designed to hold the Luma axis fixed. If we begin with what might be considered the unit Luma axis $\bar{L}_4 = \begin{bmatrix}0.5 & 0.5 & 0.5 & 0.5\end{bmatrix}^T$, it might make sense to pair it with an axis that would hold fixed the overall luminance of the mid- and side-channels separately, such as $\bar{L}_{2-2} = \begin{bmatrix}0.5 & 0.5 & -0.5 & -0.5\end{bmatrix}^T$, for construction of the 2-blade for our rotation matrix. However, the rotation method itself takes a variable vector as a parameter. Instead of hard-coding the above $\bar{L}_4, \bar{L}_{2-2}$, we normalize the parameterized $\bar{v}_{\text{param}} \rightarrow \begin{bmatrix}x & y & z & w\end{bmatrix}^T$, and calculate its wedge product with a copy of itself that has had the sign of the second two entries flipped. This 2-blade is then used with Rodrigues' rotation formula[14] to compute the 4D rotation matrix, $\boldsymbol{R}$:

$$\boldsymbol{K} = \begin{bmatrix}x\\y\\z\\w\end{bmatrix} \wedge \begin{bmatrix}x\\y\\-z\\-w\end{bmatrix} = \begin{bmatrix}0 & 0 & -2xz & -2xw\\0 & 0 & -2yz & -2yw\\2xz & 2yz & 0 & 0\\2xw & 2yw & 0 & 0\end{bmatrix} \tag{10}$$

$$\boldsymbol{R} = e^{\theta \boldsymbol{K}} \tag{11}$$

One area of modification we were not able to endeavor upon was to alter StyleGAN3's new and extensive metrics library for better

reporting and evaluation of our results. Since these metrics are highly specialized for dealing with visual images, the prospect of inventing numerous (and potentially half-baked) parallels proved infeasible for the time period allotted.

### 4.3 Training Configuration

Both the hardware and software requirements for StyleGAN3 are strict, and the training algorithm is quite demanding in terms of computing power. While this data is not presented for StyleGAN3, StyleGAN2 was trained for 25000 kimgs across 8 GPUs, taking a total elapsed time of 9d 18h[9]. Further, with the increased size of our data set (512x512 .png files would normally be 110kb each, while our ~12s .wav files produce spectrograms that are 4.2MB each), this process is elongated, and storage becomes a huge issue. We do not possess a machine capable of running StyleGAN3, so we opted to go with a paid plan for a remote GPU service through Paperspace's Gradient[12]. However, even then, getting data onto and off of the remote server became a hassle (normal upload/download options were not available due to the number and size of individual files), and so did training, as the virtual computers would get shut down after a maximum of 6 hours, even while tasks were running.

For uploading files to Gradient, we split our training set into distinct 2GB folders, uploaded them to Dropbox, and used `wget` from the remote machine to download them there. For downloading the files to our local machine, we enabled remote access to the local machine via public IP, and used `scp` from the remote machine to transfer files.

When running the training loop, we were only able to set up the virtual machine with a single RTX-5000 GPU, which also affected training time. However, when it came to the final training runs, we did discover the ability to regularly save training snapshots, and to resume training from any previously saved snapshot. Our spectrograms are also not equivalent when flipped, as the data is presented on a linear time axis, so we cannot take advantage of x-flips during training. After studying the Training Configurations document, and considering all of the above, we landed on training with the following parameters:

```
python train.py --outdir=trained_networks/europa_net2
 --gpus=1 --batch=8 --gamma=8 --data=datasets/europa.zip
 --cfg=stylegan3-t --kimg=10 --tick=1 --snap=1
 --metrics=none
```

10 kimgs was about how much we could accomplish with the full 6 hours, but we did resume the training from the final snapshot for another 10 kimgs, for a total of 20 kimg training loops. This is a far cry from the 10,000-25,000 kimgs used by NVIDIA when demonstrating StyleGAN2.

## 5 RESULTS

Timeframe restrictions aside, training was also halted after 20 kimgs as it seemed the network was diverging. After the first two rounds were complete, we exported the first 10 random-seed (0-9) images for each of the 20 snapshots taken by using the following command, replacing each snapshot's filepath and kimg amount for the destination folder in turn:

```
python gen_images.py --network trained_networks/europa_net2/
 00003-stylegan3-t-europa-gpus1-batch8-gamma8/network-snap
 shot-000010.pkl --outdir /notebooks/generated/20 --seeds 0-9
```

The images are quite striking, as can be seen in the Appendix table, though all seeds seem to tend towards brightness and monochromaticity after ~10-13 kimg training loops, which is the opposite of what we wish to see. When a standard instance of StyleGAN3 is trained on .png images, it may not produce *realistic* images after 20kimgs of training, but they are usually recognizable attempts at reproducing the target subject. However, none of these images seem to match the qualitative textures found when running audio samples from our training set through the .png transform.

The audio files produced by the trained network are quite novel, though fairly same-y in overall qualia of sound: A big, clicky, buzzy, bassy drone with lots of stereo phase shifting, layered by rising/falling/spinning frequencies. While thrilling to hear for the novelty, they do not seem usable in a musical context.

As can be seen in Fig. 3, the generated audio files have extreme DC offsets, meaning our transform/reverse-transform's assumption that the zero-frequency will be equal to zero may be hubristic. The periodicity of the generated file can also be seen as being more sparse than that of the file from the training set. This is what causes them to sound clicky, but also what makes them sound bassy, as the mind interprets the space between peaks as a subharmonic frequency. The full software package including JPH Results set (all images from the table, in 512×512 .png format, with their matching .wav files) can be downloaded here[6]. While these results are somewhat disappointing, much progress has been made, and there is still more work that could be attempted using present technology.

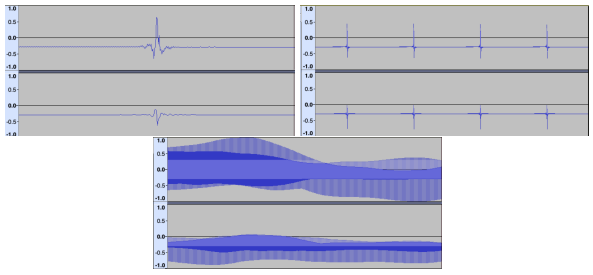
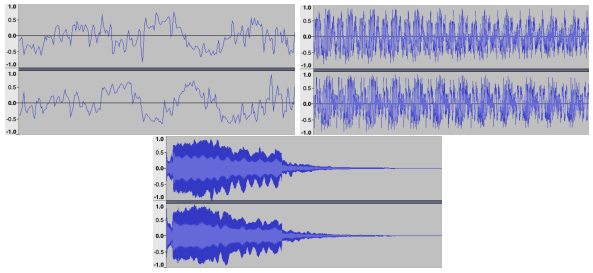


**Figure 3: Audio waveform generated by 0-seed after 02 kimg training loops, displayed at three different magnifications. (Upper-Left: ~0.05s | Upper-Right: ~0.1s | Bottom: ~12s) Below: The same, but for a file from the training set.**



## 6 FURTHER RESEARCH

It is possible that the 4D augmentation pipeline methods may not be as effective at achieving what the original 3D transformations were designed for. Specifically, it seems that the images diverged towards brightness, while the 4D rotation was intended to keep luminescence fixed. Since the objects and operations of exterior and geometric algebra required for these methods were new to us, confidence in the method is not extremely high. Further, it should be noted that the wedge product is a generalization of the cross-product to higher dimensions, so it occurred to us that using it to generate the 2-blade for the rotation matrix might have given us a 2-blade orthogonal to the two Luma axes it was designed to be parallel to. This highlights a gap in our understanding. Similarly, as it is recommended in the configurations document, we might try fine-tuning the hyperparameter `--gamma`, since gamma correction will affect an image's luminance. This might be easier to fine-tune via a standard instance of StyleGAN3 trained on a set of .png-transformed spectrograms, but the loss function should be investigated to ensure that the introduction of negative magnitudes in our lossless spectrograms won't affect the gamma hyperparameter's implementation.

One of the big improvements introduced in StyleGAN3 was improved metrics. But their breadth, and their specialization for visual images, made the prospect of modifying them daunting. Seeing what metrics improved or worsened over training time might help narrow down other problems causing divergence.

We used numpy to accomplish most of the work of the spectrogram methods, which doesn't take advantage of the GPU. This resulted in the dataset construction tool being exceedingly slow, taking over three hours for our $\sim 11,200$ item training set. These might be updated to take advantage of GPU processing by switching to pytorch, which is the package utilized by StyleGAN3.

By discarding the DC-offset information during our transform, we induced some audible low-end distortion, and as we saw, the assumption that this would be zero was upended by the results. At a sample rate of 44.1kHz, the highest frequency we could squeeze out of an RFFT is $\frac{44.1\text{kHz}}{2} = 22.05\text{kHz}$. Since the upper range of human hearing is 20kHz, and the top portion of most of the .png spectrograms from our training set are almost completely black anyway, it may make sense to instead discard the high-frequency element in order to obtain our required power-of-2 resolution.

It also may be that the spectrogram design itself is too fine-grained to be interpretable by a style-based architecture, as reflected in the broad strokes of the result images. For one, we know there is a discrete boundary for the start of an audio file, and for its upper- and lower-bound of frequency in this representation. But much of the signal processing improvements to the StyleGAN3 architecture were based on adapting them to better deal with the possibilities of what lies 'just beyond' the edge of the frame, smoothing out these and other types of discrete edges. The introduction of noise at each layer to randomize qualities like pores and hair follicles may also have a detrimental effect on the pixel-precision of this lossless spectrogram format, though hopefully the disentanglement of the input space would curb such an issue. Successive research might try to design a reversible weighted multi-windowing technique, though this may put tighter limitations on window size and audio length with the technology in its current state.

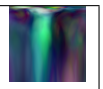
## ACKNOWLEDGMENTS

Special thanks to Dr. Aaron Tresch Fienberg, and to Nik Kanetomi and John Ahern for helping with both conceptual- and code- troubleshooting.

## REFERENCES

[1] Alex Clark and Contributors. Jun 22, 2021. Pillow: The friendly PIL fork. https://pillow.readthedocs.io/en/stable/index.html
[2] The NumPy community. Jun 22, 2021. numpy.interp. https://numpy.org/doc/stable/reference/generated/numpy.interp.html
[3] The NumPy community. Jun 22, 2021. numpy.save. https://numpy.org/doc/stable/reference/generated/numpy.save.html
[4] The SciPy community. 2022. scipy.fft.rfft. https://docs.scipy.org/doc/scipy/reference/generated/scipy.fft.rfft.html
[5] Wikipedia contributors. 19 October 2021 17:04 UTC. Spectrogram. (19 October 2021 17:04 UTC). https://en.wikipedia.org/wiki/Spectrogram
[6] Jon Henshaw. 2022. SG3-spect Software Package & Results Data Set. https://www.dropbox.com/sh/4yq1m0vfdo3al9x/AADGcJZ_TfEg3X2N7P7x6CVIa?dl=1
[7] Tero Karras, Miika Aittala, Samuli Laine, Erik Härkönen, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. 2021. Alias-Free Generative Adversarial Networks. In *Proc. NeurIPS.*
[8] Tero Karras, Samuli Laine, and Timo Aila. 2019. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.* 4401–4410.
[9] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. 2020. Analyzing and Improving the Image Quality of StyleGAN. In *Proc. CVPR.*
[10] Shuangzhe Liu, Gotz Trenkler, et al. 2008. Hadamard, Khatri-Rao, Kronecker and other matrix products. *Int. J. Inf. Syst. Sci* 4, 1 (2008), 160–177.
[11] OED Online. 2021. spectrogram, n.1. (2021). https://www.lexico.com/definition/spectrogram
[12] Paperspace. 2022. Gradient. https://gradient.run/
[13] RapidTables. 2021. CMYK to RGB color conversion. (2021).
[14] Olinde Rodrigues. 1840. Des lois géométriques qui régissent les déplacements d'un système solide dans l'espace, et de la variation des coordonnées provenant de ces déplacements considérés indépendamment des causes qui peuvent les produire. *J. Math. Pures Appl* 5, 380-400 (1840), 5.
[15] Reason Studios. 2021. Europa. (2021).
[16] Orgest Zaka, Arben Baushi, and Olsi XHOXHI. 2017. GEOMETRIC TRANSFORMATIONS IN MULTIVEC $ TORIAL LANGUAGE AND THEIR APPLICATIONS IN ROBOTICS AND ANIMATIONS. In *International Conference on Applied Sciences and Engineering. ICEAS.*

**Appendix: Evolution of First 10 Seeds Across 20 kimg Training Loops**

| Kimgs | Seed 0 | Seed 1 | Seed 2 | Seed 3 | Seed 4 | Seed 5 | Seed 6 | Seed 7 | Seed 8 | Seed 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 01 |  |  |  |  |  |  |  |  |  |  |
| 02 |  |  |  |  |  |  |  |  |  |  |
| 03 |  |  |  |  |  |  |  |  |  |  |
| 04 |  |  |  |  |  |  |  |  |  |  |
| 05 |  |  |  |  |  |  |  |  |  |  |
| 06 |  |  |  |  |  |  |  |  |  |  |
| 07 |  |  |  |  |  |  |  |  |  |  |
| 08 |  |  |  |  |  |  |  |  |  |  |
| 09 |  |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |  |  |  |
| 13 |  |  |  |  |  |  |  |  |  |  |
| 14 |  |  |  |  |  |  |  |  |  |  |
| 15 |  |  |  |  |  |  |  |  |  |  |
| 16 |  |  |  |  |  |  |  |  |  |  |
| 17 |  |  |  |  |  |  |  |  |  |  |
| 18 |  |  |  |  |  |  |  |  |  |  |
| 19 |  |  |  |  |  |  |  |  |  |  |
| 20 |  |  |  |  |  |  |  |  |  |  |