Evolving Glitch Art

Eelco den Heijer

Objectivation B.V., Amsterdam, The Netherlands
Faculty of Sciences, VU University Amsterdam, The Netherlands
eelcodenheijer@gmail.com
http://www.few.vu.nl/~eelco/

Abstract. In this paper we introduce Glitch art as a new representation in Evolutionary Art. Glitch art is a recent form of digital art, and can be considered an umbrella term for a variety of techniques that manipulate digital images by altering their digital encoding in unconventional ways. We gathered a number of basic glitch operations and created a 'glitch recipe' which takes a source image (in a certain image format, like jpeg or gif) and applies one or more glitch operations. This glitch recipe is the genotype representation in our evolutionary GP art system. We present our glitch operations, the genotype, and the genetic operators initialisation, crossover and mutation. A glitch operation may 'break' an image by destroying certain data in the image encoding, and therefore we have calculated the 'fatality rate' of each glitch operation. A glitch operation may also result in an image that is visually the same as its original, and therefore we also calculated the visual impact of each glitch operation. Furthermore we performed an experiment with our Glitch art genotype in our unsupervised evolutionary art system, and show that the use of our new genotype results in a new class of images in the evolutionary art world.

1 Introduction

Glitch Art originates from an electronic music niche called 'Glitch' [4]. Originally, a 'glitch' refers to a false electronic signal that has been caused by a short, unexpected surge of electric power (in this context, a glitch is a 'spike'). Glitch music is created using electronic instruments that have been altered in a process called 'circuit bending', whereby electronic parts are removed or short-circuited. Other forms of glitch music originate from a variety of techniques that are labelled 'data bending', taken from the hardware equivalent 'circuit bending'. In data bending, digital data is manipulated in unexpected ways to create surprising, novel output. The idea of altering a digital component to influence the analogue output soon travelled from the music domain to the visual domain. Visual glitch art also uses 'data bending' whereby artists and programmers use hex editors to open digital images, alter the binary content (often at random), save the result and view the visual effect. A popular use of glitch art is the 'Wordpad effect', whereby one opens a digital image in Microsoft Wordpad (a simple word processor). Wordpad assumes the content is a text document and will try to re-arrange the “text”, insert line endings, and replace a number of characters with other characters. All these changes may, or may not, act like ‘glitches’ in the resulting image. There are few scientific publications on the topic of Glitch art, but there are some very useful online tutorials[1]. Several authors have suggested that the name ‘Glitch’ is a misnomer, since many glitch artists deliberately manipulate digital content, and do not rely on accidental errors, or glitches [8,9]. In our paper we present a number of basic glitch operations that alter the binary encoding of digital images. We use these operations to construct a genotype, and with this genotype we perform an experiment with our unsupervised evolutionary art system.

Our research questions are

1. Is it possible to develop a genotype for Glitch art (including the operators for initialisation, crossover and mutation)? And if so, what are the main obstacles?
2. A glitch operation can ‘break’ the image, and make it unreadable. Is it possible to control the ‘fatality rate’ of glitch operations at various conditions, using various image formats?
3. A glitch operation may change a source image, but may also leave the source image unchanged from a visual point of view; is it possible to control the visual impact of glitch operations?
4. Does the evolution of glitch art contribute to the visual range of evolutionary art? In other words, can we evolve aesthetically pleasing images that are different from images that we know from existing evolutionary art systems?

The rest of the paper is structured as follows; first we discuss related work in Section 2. Next, we discuss glitch art in Section 3. In Section 4 we describe our experiments and their results. We end this paper with conclusions and directions for future work in Section 5.

2 Related work

Our paper describes Glitch as a genotype representation in Evolutionary Art. In this section we will shortly describe other genotype representations in EvoArt, and we will shortly describe related work in generating glitch art.
Evolutionary art is a field where methods from Evolutionary Computation are used to create works of art. Good overviews of the field are [18] and [2]. Matthew Lewis presents a nice overview of many forms of visual evolutionary art in the first chapter of ‘The Art of Artificial Evolution’ [10,18]. There are several methods in EvoArt that create images from ‘scratch’, like the well-known symbolic expressions [11,19,6] and (shape) grammars [10]. In recent years there have been papers on using vector graphics [7,3]. In addition, there have been investigations in the manipulation of existing images. [5] describes an approach that using non-photorealistic rendering or NPR to produce synthetic oil paintings from images;

[1] An overview of online Glitch tutorials can be found at http://danieltemkin.com/Tutorials/

the author uses a genetic algorithm to find suitable values for his NPR system. In [17] the authors describe the evolution of a NPR system using genetic programming, whereby the authors use a number of image filter primitives. Our glitch approach is somewhat similar to [5] and [17] in the use of a source image as a starting point. However, our glitch approach does not model any form of filtering or NPR approach, so the visual output of our approach is rather different; our approach often results in images with more displacement and distortion.
Glitch art is a very new field within the digital art world, and although there have been numerous small projects and many DIY enthusiasts that have created and uploaded glitch images (for example, search Flickr or Google images for 'Glitch'), there have been very little scientific publications on the subject. Ben Baker-Smith has created a software program called *GlitchBot*[2] that daily selects images (with a Creative Commons license), applies a glitch operation on them and posts the result to Flickr Glitch Art pool[3]. GlitchBot searches a random character in the image data and replaces it with another character. If the image 'breaks', the system repeats the process, until a valid image is created [1]. Manon and Temkin have published a collection of notes on Glitch art [12]. A good art theoretical reference on visual glitch art is [15]. A good starting point with many visual examples is [16]. Next to music and visual arts, the 'Glitch' phenomenon has moved to animation [20] and even literature [13].

3 Glitch Art

Glitch art and evolutionary art share a number of similarities. Both employ a sort of 'generate and test' paradigm, whereby a software program generates a number of possibilities, and a selection is performed by an artist or by a software component. Manon et al state that one can not create an glitch image, one can merely *trigger* a glitch, and this volatile nature of glitch art makes it a pseudo-aleatoric art form [12]. Applying a glitch operation to an image is very simple, but creating interesting visual content is far from trivial. As Manon et al state "Glitch art is like photography; it's easy to do, but it's hard to do well" [12]. Although finding interesting visual content using Glitch is difficult, it is by no means a random process. Applying the same glitch operations on the same image will result in the same end image. In our EvoArt system we support six image file types for Glitch art; Windows Bitmap (bmp), gif, jpeg, raw (uncompressed raw image data), png and (compressed) tiff. The image formats each have their own binary format, and each format has its characteristics with respect to glitch operations. First, uncompressed data formats (raw, bmp) are 'more stable' than compressed formats under glitch operations. Glitch operations on these types of images will affect image data, whereas glitch operations on compressed image data formats might affect meta-data that contains instructions on the compressed format. The probability of making an image unreadable for

[2] http://bitsynthesis.com/glitchbot/
[3] http://www.flickr.com/groups/glitches/pool/

image viewing software is higher when using compressed image formats such as png, jpeg and gif.

3.1 A genotype for Glitch Art

Glitch art is process art; one does not create a glitch, one *triggers* a glitch [12]. In our EvoArt system (using GP) we want to follow up on this idea, and evolve 'glitch recipes'. A glitch recipe starts with a randomly selected source image, and applies one or more glitch operations. The phenotype is the resulting 'glitched' image. We implemented operations for the insertion of a random byte string, the removal of a part of the binary image, and the replacement of a byte with another byte. These operations 'insert', 'delete' and 'replace' are typical examples of manual glitch art; you could easily perform these with a hex editor. Since we are performing these operations automatically in software, we also added a number of operations that are easily done by software, but would be difficult to perform manually. These are the binary operations 'and', 'or', 'xor' (exclusive or) and 'not'. Furthermore, we added a 'reverse' operator that randomly reverses a number bytes from a certain position. The context of all binary operations is the binary image format, and plays a very important role in visual glitch art. As described in the previous section, image formats vary in their layout and content. If you take a JPEG image and convert it to BMP format, the binary encoding is different. Therefore, if you perform a random operation f and perform it on either a JPEG or a BMP image, the results will be different; it might be that the operation has no effect on either image, but if it does have an effect, it will probably be different from a visual point of view. It is entirely possible that the operation f is destructive on the JPEG image and not on the BMP image, or vice versa. Since the image format is important, we have added a 'setImageFormat' operation that changes the binary encoding within the genotype. The genotype starts with reading its source image, and the binary encoding will be the one of the source image. Executing the 'setImageFormat' operation will save the source image, plus all applied glitch operation so far (if any) and converts the 'current' image format to the new specified image format.
Table 1 gives an overview of the glitch operations in our EvoArt system. Several glitch operations from Table 1 use 'position' and/ or 'size' as an argument. Both are relative numbers in $[0 \ldots 1]$ where the actual position is calculated at runtime. The 'relative' argument position or size is multiplied with the image size to obtain the absolute position or size. This abstraction makes the operation independent of image size, and makes it easier to transfer an operation from one genotype to another by crossover. The position arguments are initialised between 0.02 and 1.0; we chose 0.02 in order to avoid touching the first 2% of the binary encoding, where several image formats store 'delicate' metadata; touching this metadata often results in immediate destruction of the image. The threshold of 2% was chosen after a number of trials; further experiments should determine more elaborate thresholds, we suspect that different image file formats will have different thresholds. The 'size' arguments, used in the 'delete' and 'not' operation specifies a relative size between 0 and 1. The size arguments

Operation	Argument 1	Argument 2	Description
insert	*position*	*random bytes* of length N ($N \in [2, \ldots, 64]$)	Inserts random bytes at a certain position
delete	*position*	*size*	deletes N bytes from a certain position, where $N = size \cdot imagesize$
replace	*byte1*	*byte2*	replaces every occurrence of *byte1* with *byte2*
and , or, xor	*position*	*bit mask* of length N ($N \in [2, \ldots, 64]$)	Performs a binary operation at a certain position using the bitmask
not	*position*	*size*	inverts N bytes starting at a certain position, where $N = size \cdot imagesize$
reverse	*position*	*size*	reverses N bytes from a certain position, where $N = size \cdot imagesize$
setImageFormat	[png|gif|jpg|tiff|raw|bmp]	-	Saves the current image in the specified format, and reads the binary data from the new format

Table 1: The glitch operations used in our experiments. arguments of type *position* are in $[0.02 \ldots 1]$; the actual position is calculated by multiplying the *position* argument with the size of the uncompressed image (see Section 3.1 for a further explanation).

are initialised between 10^{-4} and 10^{-2}, and for an 50kb image the absolute size will lie between $10^{-4} \cdot 50 \cdot 1024 \approx 5$ and $10^{-2} \cdot 50 \cdot 1024 \approx 512$ (so a 'reverse' operation on a 50kb image will randomly reverse a buffer between 5 and 512 bytes). In Figure 1 we give a number of examples of the glitch operations and their results; we show the portrait of computer graphics celebrity Lenna[4], and 7 glitch operations in several image formats. In addition, Figure 1 shows two examples of two glitch programs each containing 4 glitch operations.

3.2 Initialisation

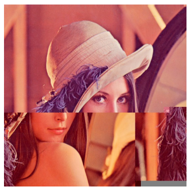
Our initialisation procedure randomly samples a source image from a specified image directory and in our experiments we created a image test set of 500 images. Next, the initialisation creates between 1 and 5 glitch operations.

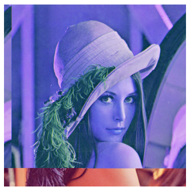
3.3 Crossover

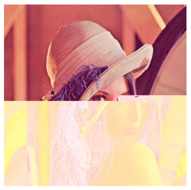
The implementation of the two-parent crossover for our glitch genotype is fairly straightforward. First, the crossover randomly selects the source image from one

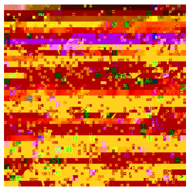
[4] The image of Lenna has been used as an example image in many scientific papers, especially in the computer graphics community. Also see http://en.wikipedia.org/wiki/Lenna

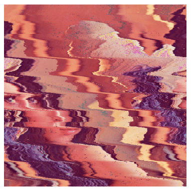
(a) orginal

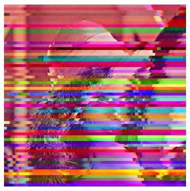

(b) delete/raw

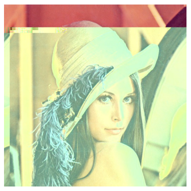

(c) insert/bmp

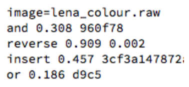

(d) reverse/jpeg


(e) replace/jpeg


(f) replace/gif

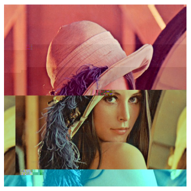

(g) replace/tiff

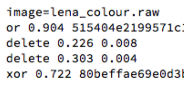

(h) xor/jpeg



```
image=lena_colour.raw
and 0.308 960f78
reverse 0.909 0.002
insert 0.457 3cf3a147872
or 0.186 d9c5
```

(i) example 1


(j) example 1



```
image=lena_colour.raw
or 0.904 515404e2199571c
delete 0.226 0.008
delete 0.303 0.004
xor 0.722 80beffae69e0d3
```

(k) example 2

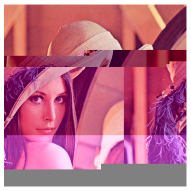
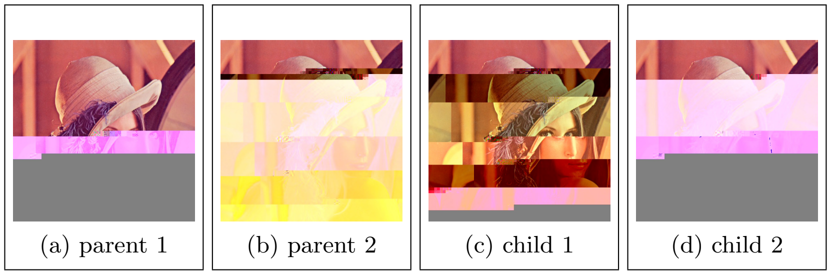

(l) example 2

Fig. 1: Several examples of glitch operations. Every image is the result of one glitch operation on the Lenna image in a certain image format; the captions show the operation and image source format. The bottom row (1i-1l) shows two examples of genotypes with 4 glitch operations each, with the resulting image/ phenotype.

of the parents. Next, the list of of glitch operations of both parents are cut in two, and a new list is created by concatenating the first half of one randomly selected parent with the second half of the other parent. Figure 2 shows an example of a crossover operation on two Glitch programs.

3.4 Mutation

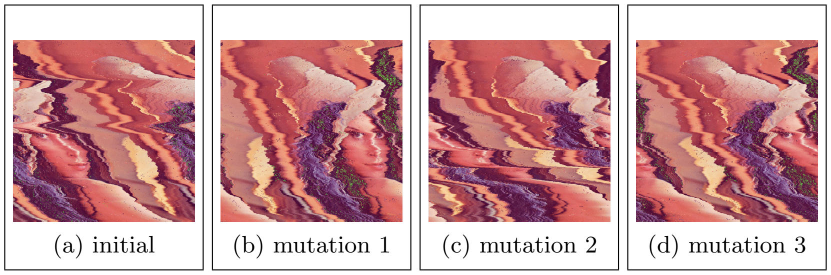
The mutation operator acts on all parts of the genotype. It may alter the source image by choosing a random new image (with a probability of 0.1). It iterates over the glitch operations, and replaces an existing glitch operation with a random new one (with a probability of 0.1), or alters an existing one by changing the arguments of the operator. For numeric arguments, it adds or subtracts a value within 1% of the original value. For byte arrays, it iterates over all the bytes and replaces a byte with a random new byte (with a probability of 0.01).



(a) parent 1 (b) parent 2 (c) child 1 (d) child 2

Fig. 2: Examples of a crossover operation; the first image two images are the two parents, each consisting of the Lenna image as the source, and 5 random glitch operations. The right two images are the results of crossover.

For single byte arguments (of the 'replace' operator) it increases or decreases the byte value with a value between 0 and 4 (thereby clamping the resulting byte value between 0 and 255). Figure 3 gives a few examples of three mutations of one individual glitch program.



(a) initial (b) mutation 1 (c) mutation 2 (d) mutation 3

Fig. 3: Examples of mutations; the first image is the initial glitch program; a program consisting of a singe 'replace' operation on a gif image. The other three images are three mutations of the initial glitch program.

4 Experiments

In this section we describe our experiments with Glitch art. In our first experiment we determine the fatality rate of our glitch operations. The fatality rate is calculated as the number of broken images divided by the total number of glitch operations. In the second experiment we calculate the visual impact of our glitch operations, whereby we measure the average amount of visual change each glitch operation causes. In the third experiment we evolve glitches images using our glitch genotype without a human in the loop.

4.1 Experiment 1: determining fatality rate

In our first experiment we calculate the fatality rate of the glitch operations and the different image formats. If we apply a random glitch operation on a random image of a certain image format, there is a probability that the resulting image will be broken (i.e. invalid). From present literature, little is known about the probabilities per image format or per glitch operation. Therefore, we decided to measure the fatality rate per glitch operation. To this end, we created an image set of colour 100 images from various sources (mainly paintings, news photos, pictures of cats, etc.). We converted all 100 images to all our six supported image formats (bmp, gif, jpeg, png, raw, tiff) using ImageMagick. Next, for each image we created a random glitch program consisting of one random glitch operation. We applied this glitch operation on the source image, and determined whether the resulting image was 'valid' (i.e. not broken). We repeated this 10 times for each image format, resulting in 1000 calculations per operation-format combination. We measured the number of broken images, and divided this by the total number of glitch operations. The results are shown in Table 2. From

	bmp	gif	jpeg	png	raw	tiff	
insert	0.000	0.168	0.007	0.997	0.009	0.998	0.363
delete	1.000	0.166	0.006	1.000	0.007	1.000	0.530
replace	0.018	0.180	0.120	0.996	0.101	0.198	0.269
and	0.000	0.016	0.002	0.997	0.000	0.010	0.171
xor	0.000	0.024	0.007	0.998	0.007	0.014	0.175
or	0.000	0.013	0.145	0.999	0.145	0.149	0.242
not	0.005	0.610	0.310	1.000	0.277	0.650	0.475
reverse	0.006	0.124	0.094	1.000	0.115	0.436	0.296
	0.129	0.163	0.086	0.998	0.083	0.432	

Table 2: The results of the calculation of the fatality rate of each glitch operation per image file format. Each number is the average of 1000 calculations. The bottom row shows the averages per image file format, and the rightmost column shows the averages per glitch operation.

the results in Table 2 we can conclude that png is by far the most 'sensitive' image format, since it has the highest fatality rate. Its fatality rate is almost 1.0 (100%) for any glitch operation, from which we may conclude that png is rather unusable as an image format for glitch operations. The uncompressed format 'raw' has a very low fatality rate. The Windows Bitmap format also has a relatively low fatality rate, but it does have a 100% fatality rate with 'delete' operations. Gif, bmp, jpeg and raw have low fatality rates, and we will restrict future glitch experiments to these image formats. In our experiment tiff scored high on fatality rate, and we suspect that this is caused by our use of compressed tiff images. We think that when we use uncompressed tiff images (tiff is a very versatile image format, and support both compressed and uncompressed data), tiff will score similar to the raw format on fatality rate. When we focus on the glitch operations in Table 2 we see that the 'delete' operation is the most 'destructive' glitch operation, with a fatality rate of 0.530 (53%). The score is especially high since three image formats do not 'work well' with random deletions of bytes; bmp, png and compressed tiff all score 1.0 (100%) on 'delete' operations. The ‘not’ and ‘insert’ operation also have a high fatality rate with average scores of 0.475 and 0.363 respectively.

4.2 Experiment 2: measuring visual impact

Although it is interesting to know the fatality rate for each glitch operation and each image format, it is also interesting to know the average visual impact of each glitch operation per image format. We loosely define visual impact as the difference between the resulting ‘glitched’ image and its source image. From our first experiment we know that uncompressed image formats (most notably ‘raw’) are more ‘resistant’ to glitch operations than several compressed image formats (most notably ‘png’), but does that also mean that glitch operations have less visual effect on uncompressed image formats? To verify this, we did an experiment similar to our first experiment, but instead of measuring the fatality rate, we measured the visual impact. We calculate the visual impact as follows; we start with the source image I_a, apply one of the glitch operations from Table 1 and obtain the ‘glitched’ image I_b. We convert I_a and I_b to grayscale images, and calculate the distance between the two images by calculating the average difference in grayscale value.

$$d_{grayscale}(I_a, I_{b)} = \frac{\sum_{x=0}^{x<w} \sum_{y=0}^{y<h} |I_a(x, y) - I_b(x, y)|}{w \cdot h} \tag{1}$$

where $I_x(x, y)$ represents the grayscale value of the pixel at (x, y), and w and h are the width and height of the images (images a and b have the same width and height). We calculated the visual impact for each combination of glitch operation and image format on a test set of 100 images (the same image set as used in the first experiment), and performed 10 runs (resulting in 1000 calculations per operation/ format combination). If a glitch operation results in a broken image, we can not calculate the grayscale distance, and we return the value 0. The results are presented in Table 3.

	bmp	gif	jpeg	png	raw	tiff	
insert	0.0000096	0.0014833	0.0002334	0.0000001	0.0010734	0.0000000	0.0004666
delete	0.0000000	0.0014758	0.0002661	0.0000000	0.0012873	0.0000000	0.0005049
replace	0.0000155	0.0204895	0.0020033	0.0000000	0.0023181	0.0007292	0.0042593
and	0.0000000	0.0006621	0.0001791	0.0000000	0.0009455	0.0000057	0.0002987
xor	0.0000000	0.0005464	0.0002195	0.0000000	0.0010731	0.0000012	0.0003067
or	0.0000000	0.0004249	0.0001382	0.0000000	0.0009570	0.0000094	0.0002549
not	0.0000042	0.0005932	0.0001423	0.0000000	0.0007850	0.0000061	0.0002551
reverse	0.0000003	0.0014581	0.0001809	0.0000000	0.0010178	0.0000098	0.0004445
	0.0000037	0.0033917	0.0004203	0.0000000	0.0011821	0.0000951	

Table 3: The results of the calculation of the visual impact (or image distance) of each glitch operation per image file format. Each number is the average of 1000 calculations. The bottom row shows the averages per image file format, and the rightmost column shows the averages per glitch operation.

From our second experiment we can conclude that glitch ‘gif’ and ‘raw’ result in the largest visual changes. From the first experiment we concluded that ‘png’ is a very difficult image format for glitch operations (since most glitch operations result in a broken image), and this results in an average of 0.0 for the grayscale distance (since we assume $d_{grayscale} = 0$ in case of a broken image). The ‘replace’ operator has the highest visual impact, which confirms our presumptions after several manual experimentations with a hex editor. Note that the aforementioned GlitchBot uses the ‘replace’ operator exclusively.

4.3 Experiment 3: Unsupervised Evolutionary Art

With our genotype, our initialisation, crossover and mutation we performed 20 runs of unsupervised evolution with a population of 100, a tournament size of 2 and 10 generations per run. We used 500 gif images of famous paintings as the pool for the source images (the individuals in the population sample a random image from this pool). We used a simple ad hoc aesthetic measure that resembles the Global Contrast Factor (or GCF) aesthetic measure. The GCF aesthetic measure calculates contrast at various resolutions in the image; images with low contrast are considered ‘uninteresting’ and receive a low score. For more details we refer to the original paper [14]. Our aesthetic measure does not calculate the difference in intensity (contrast) but the difference in colour/ hue. We realise that this measure would favour phenotypes in our system that have source images that already score high on this measure, which means that this measure is not specifically tailored for glitch operations. A measure that would be tailored for glitch operations would at least calculate the difference between the glitched images and the source images. We intend to develop a custom aesthetic measure for Glitch art, and combine this new aesthetic measure with existing aesthetic measures in a Multi-objective EA setup in future work. Figure 4 shows the results of 10 images from our unsupervised runs. Note that the first two images result from the same source image, and the same goes for image 3,4 and 5. We think that the visual output over 20 runs is varied, although a number of individual runs contained images that were relatively similar. Since our primary goal in our experiment was to test the new genotype and its genetic operators, we kept our EA as standard as possible, and did not use any population diversity strategy.
During our runs we measured the glitch operation frequency in the individuals in the populations. After 20 runs of 10 generations, the ‘replace’ operation was most frequent, with a score of 27% (which means the 27% of all glitch operations in all individuals in the population is the ‘replace’ operation. Note that a ‘replace’ operation can occur multiple times in the same individual glitch program). The ‘delete’ and ‘insert’ operation occur least frequent, with a frequency of 7.8% (delete) and 7.1% (insert). We suspect that the fatality rate of a glitch operation act as a negative selection pressure, since a broken image results in a fitness of -1. We also measured the fatality rate of the individuals in the population, and this fatality rate varied between 0.13 and 0.2 (13% - 20%).

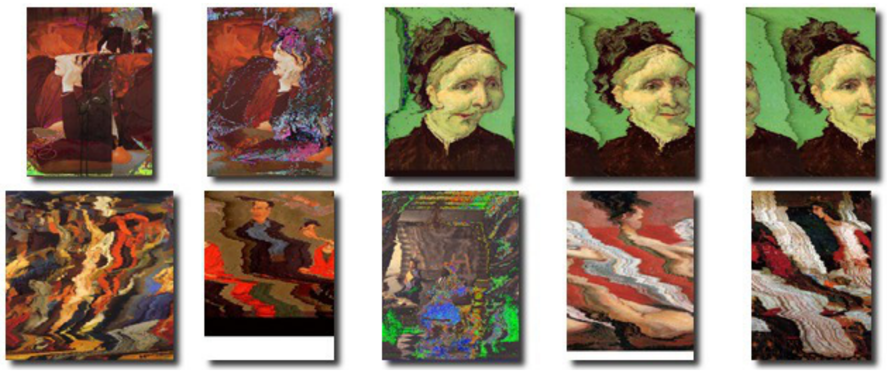


Fig. 4: Portfolio of images gathered from twenty runs with our Glitch genotype and genetic operators, using our Colour Contrast (hue) aesthetic measure.

5 Conclusions and Discussion

In Section 1 we presented a number of research questions and we will answer them here. First, we asked whether it was possible to evolve Glitch Art using a new genotype for Glitch. Our experiments with Glitch art confirm this. We have developed a genotype for Glitch art and have implemented an initialisation, crossover and mutation operator, and have performed unsupervised evolution with these new genetic operators. Our main obstacles were the high fatality rates of some glitch operation/ image format combinations, and the lack of robustness of some image decoding libraries; we encountered a number of crashes when trying to read invalid image content.
Our second research question involves the control of the fatality rate of glitched images; from our first experiment we can conclude that the choice of image format and glitch operation has a large effect on the fatality rate. Using glitch operations on png images will result in broken images in almost 100% of the cases, which makes it an unusable image format for glitch operations. From our fatality rate calculation in Section 4.1 we concluded that we should restrict glitch operations to the image formats gif, jpeg and raw.
Our third research question was whether we could control the the visual impact of the glitch operations. We have measured the visual impact of the different combinations of glitch operation and image format, and found that gif and raw produced the most visual changes upon glitch operations. With the results of the first and second experiment, we concluded to use gif as the image format for our third experiment. We intend to use the numbers from experiment 1 and 2 to decrease the fatality rate and increase the visual impact of our glitch system. Nevertheless, creating glitch art is, and will be, a trial-and-error process.
Our last research question was whether our experiments with Glitch art resulted in a style of images that is 'new' within evolutionary art. Although it difficult to answer this question in quantitively, we think that glitch images differ significantly from most evolutionary art images; the images have a more 'radical'

flavour than images evolved with image filters, since there is a higher level of displacement and distortion in the 'glitched' images.

References

1. Ben Baker-Smith. Personal communication.
2. P. J. Bentley and D. W. Corne, editors. *Creative Evolutionary Systems.* Morgan Kaufmann, San Mateo, California, 2001.
3. Steven Bergen and Brian Ross. Automatic and interactive evolution of vector graphics images with genetic algorithms. *The Visual Computer*, 28:35–45, 2012.
4. Kim Cascone. The aesthetics of failure: "post-digital" tendencies in contemporary computer music. *Computer Music Journal*, 24(4):12–18, dec 2000.
5. John Collomosse. Evolutionary search for the artistic rendering of photographs. In Romero and Machado [18], pages 39–62.
6. E. den Heijer. Evolving art using measures for symmetry, compositional balance and liveliness. In *Proceedings of the 4th IJCCI 2012*, pages 52–61, Barcelona, Spain, 2012.
7. E. den Heijer and A. E. Eiben. Evolving pop art using scalable vector graphics. In *EvoMusart 2012, Evolutionary and Biologically Inspired Music, Sound, Art and Design, LNCS 7247*, pages 48–59, Malaga, Spain, 2012.
8. Jonas Downey. Glitch art. *Ninth Letter*, 2002.
9. Duncan Geere. Glitch art created by 'databending'. *Wired Magazine*, 2010.
10. Matthew Lewis. Evolutionary visual art and design. In Romero and Machado [18], pages 3–37.
11. Penousal Machado and Amílcar Cardoso. All the truth about nevar. *Applied Intelligence*, 16(2):101–118, 2002.
12. Hugh S. Manon and Daniel Temkin. Notes on glitch. *World Picture*, 6, 2011.
13. Stacey Mason. Glitched lit: possibilities for databending literature. In *Proceedings of the 2nd workshop on Narrative and hypertext*, NHT '12, pages 41–44, New York, NY, USA, 2012. ACM.
14. Kresimir Matkovic, László Neumann, Attila Neumann, Thomas Psik, and Werner Purgathofer. Global contrast factor-a new approach to image contrast. In László Neumann et al, editor, *Computational Aesthetics*, pages 159–168. Eurographics Association, 2005.
15. Rosa Menkman. *The Glitch Moment(um)*, volume 04 of *Network Notebooks*. Institute of Network Cultures, Amsterdam, 2011.
16. I. Moradi, A. Scott, J. Gilmore, and C. Murphy. *Glitch: Designing Imperfection.* Mark Batty Publisher, 2009.
17. Craig Neufeld, Brian Ross, and William Ralph. The evolution of artistic filters. In Romero and Machado [18], pages 335–356.
18. Juan Romero and Penousal Machado, editors. *The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music.* Natural Computing Series. Springer Berlin Heidelberg, November 2007.
19. Karl Sims. Artificial evolution for computer graphics. *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, 25(4):319–328, July 1991.
20. Mark Tribe, Reena Jana, and Uta Grosenick. *New Media Art.* Taschen, 2006.