

Generation of Vector-Based Graphics from Existing Bitmap Images by Means of the Genetic Algorithm

Chris Weller

chris_weller@hp.com

924 Shire Ct. Fort Collins, CO 80526

(970) 206-9831

Abstract

The generation of vector-based graphics from bitmap graphics is a task well suited for the search capabilities of the genetic algorithm. Vector-based graphics, which describe the position and color of geometric primitives on a plane, are typically much smaller in size than a similar bitmap image would be. Because of their small size, it is possible to employ the genetic algorithm to search through the space of all possible arrangements of those geometric primitives in order to find a combination that visually represents an existing bitmap image. Although the resulting vector-based graphic will not be identical in appearance to the original image, the new representation of the image will receive the benefits of small size and clean scaling that vector-based graphics possess.

Introduction

The encoding of images in digital formats has been an important area of study to date as the use of digital imaging has become very common through the ever-increasing popularity of the Internet, digital photography, and computers in general. Improved image encoding techniques allow for more efficient storing and transmission of digital images for viewing, analysis, and printmaking purposes which enables people to get more out of their network bandwidth, cpu, and storage capabilities.

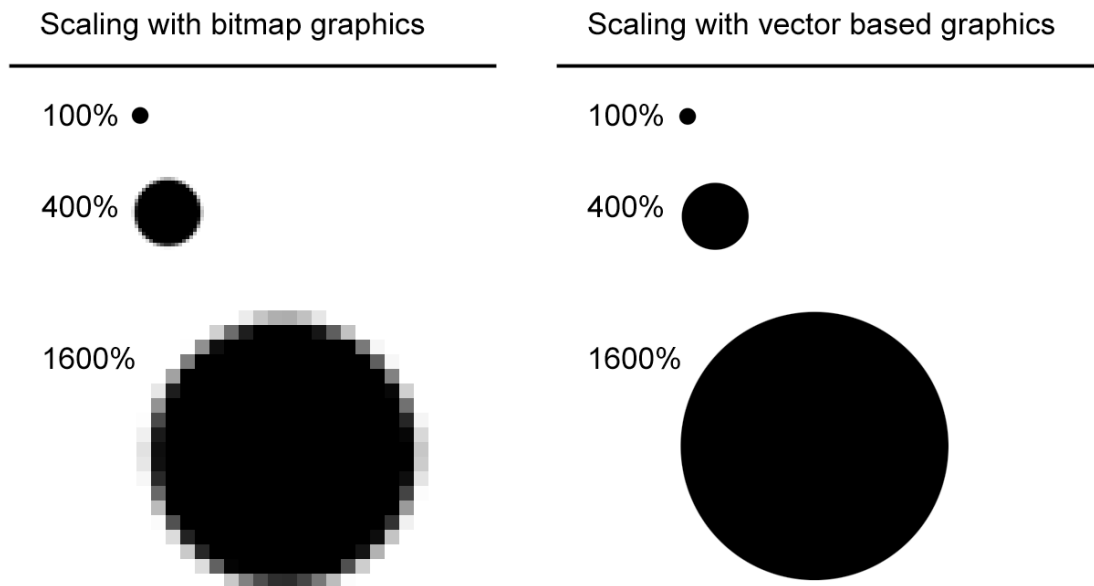
One method of representing an image involves organizing a number of flat shaded, geometric primitives such as polygons, ovals, lines, and wedges, so that they come together to construct an image or figure of some sort. Although these representations of images, called vector-based graphics, often do not contain the subtle color gradients and soft edges of photographs, they do offer some benefits over more traditional encoding techniques which attempt to store the color of each individual pixel in the original bitmap image.

One benefit of using vector-based graphics is their very small size. Another is their ability to scale in size without ugly pixelation artifacts emerging as shown in figure 1. Vector based graphics are popular formats for clipart images that come with word processing software (because of their ability to scale cleanly to any size) and with some web graphics software (because of their small size).

The generation of scalable vector-based images from existing digital images is a problem well suited for the search capabilities of the genetic algorithm. The search space in which the genetic algorithm works contains all possible arrangements, sizes, shapes, and colorings of a number of geometric shapes. Different points in the space each create different images from the composition of shapes specific to that location. By generating a fitness for each individual in the GA based on how similar it is to the target image, the population of the GA will eventually converge on areas in the search space, which contain individuals that closely represent the target image.

This paper describes the design and implementation of a software program that creates a vector-based image from an existing bitmap image by employing the genetic algorithm to orient and color a fixed number of polygons in a way that makes them visually resemble the target bitmap image.

Figure 1. Bitmap vs. vector-based graphic scaling comparison



Search Space

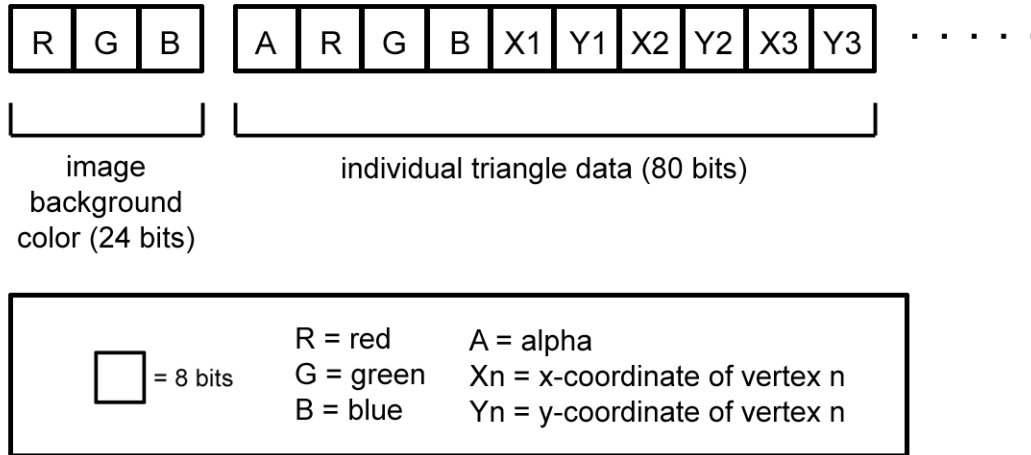
The search space in which the genetic algorithm described in this paper runs includes all possible arrangements and shadings of T triangles as well as all possible 24-bit background colors. Each individual's genetic data is $3 + 10 * T$ bytes long. The first three bytes of data represent the red, green, and blue intensities of the image's background color, respectively. After the initial 3 bytes, each 10 bytes thereafter holds the position and shading information for a different triangle. The organization of each triangle's 10-byte data structure is illustrated in figure 2. Each triangle contains four bytes describing its shading (red, green, blue, and alpha) and six bytes describing its position ($x1, y1, x2, y2, x3, y3$). Since completely transparent triangles are possible and occlusion of one triangle by another is possible, there will be many different genotypes in the search space which generate the exact same solution to the problem.

The size of the search space for a given run is exactly $256^{(3 + 10 * T)}$. For a run with $T=20$ triangles, there would be 256^{203} or approximately $7.5 * 10^{488}$ possible individuals in the search space. The size of this search space is enormous, as it should be, to be able to provide an approximation of any image you supply the GA with. With a search space of this magnitude, an exhaustive or inefficient search would be completely pointless to run as a good solution would not be found for a very, very long time. The search space is all completely accessible to the GA since the genotype is simply a fixed length binary string and the genetic operators can change any bit to any value.

The schemata for this problem describe different combinations of shading and positioning information for the triangles and background color in this problem. A schema that only contains information about one well placed, and colored triangle, will most likely rise above the average as it describes a very good trait for a solution to have. Another schema might have the upper bits of the blue channels for several triangles and the background turned on. This schema would do well if the target image contained a lot of blue. The number of schemata in this problem is $3^{(8 * (3 + 10 * T))}$. Although most of the schemata will never be seen in any generation of a given run, the better ones will start to appear as they reproduce throughout the population and then better variations take their place.

Figure 2. Organization of genetic data

Organization of genetic information



Fitness

For each run of the genetic algorithm, a target image is specified. This target image represents what the vector-based graphic solution of the GA run should strive to look like. This target image's pixel shadings (one byte red, green, and blue intensities for each pixel) are designated as the fitness cases for the run. In order to determine the fitness of an individual, the vector-based graphic described by the genetic material of an individual has to be rendered into a bitmap image with the same pixel dimensions as the target image. For each pixel in the target image, the corresponding pixel in the solution's image is compared and the sum of the square of the differences between the two pixels' red, green, and blue intensities are accumulated for every fitness case (pixel of the image). This method of fitness determination rewards solutions similar to the target image with lower fitnesses than those that are not as similar. The squaring of the difference between the color channels' intensities creates an exponentially greater penalty the further the two colors are from each other. This is meant to place more emphasis on getting the boundaries of color regions correct. Any solution with a region that differs greatly in color from the corresponding region in the target image is more severely penalized than an individual solution that differs by just a small amount from many of the fitness cases.

After the raw fitness is determined, a fitness scaling operation, which will be described in the next section, is used to normalize the fitness values of all the individuals so that fitness proportionate selection will more effective. Because of the incredibly high fitness values achieved using larger target images, all of the individuals would have nearly the same fitness, proportionately, if it were not for the fitness scaling function.

Implementation

Custom software was coded to run the genetic algorithm on this problem. The software handled all aspects of the run from the rendering of vector-based graphic images, the fitness determinations, and the actual implementation of the genetic algorithm (see table 1).

Table 1. Genetic Algorithm Tableau

Objective:	Create a vector-based image consisting of T triangles of varying color, transparency, and position, which create an image that visually resembles a target bitmap image.
Representation Scheme:	T = number of triangles to compose vector image with Structure = Binary string of $8 * (3 + 10 * T)$ bits Mapping : byte[0,1,2] = background red, green, blue byte[3+10n -> 12+10n] = Triangle n information
Fitness Cases:	All pixels' red, green, and blue intensities from the target image
Fitness:	$(255 * 255 * 3 * \text{target_width} * \text{target_height}) -$ sum over all x, y of : $(\text{target}[x][y].\text{red} - \text{solution}[x][y].\text{red})^2 +$ $(\text{target}[x][y].\text{green} - \text{solution}[x][y].\text{green})^2 +$ $(\text{target}[x][y].\text{blue} - \text{solution}[x][y].\text{blue})^2$
Standardized Fitness:	$\text{raw_fitness} - \text{worst_raw_fitness} + ((\text{best_raw_fitness} - \text{worst_raw_fitness}) / 9)$
Parameters:	M = 200 G = 10000+
Termination Criteria:	Generation == pre-designated last generation
Result designation:	Most fit individual from the last generation of the run

The initial population of the run is randomized by setting each byte in the individual's genetic data to a random value from 0 to 255. The rendering of each individual's vector-based solution to a corresponding bitmap image (figure 3) is accomplished by initializing a scratch image to the same size as the target image and filling it with the background color specified by the first three bytes of the individual's genetic data. From there the T triangles are rendered in the order that they appear in the genetic data.


For each triangle, the bytes x1, y1, x2, y2, x3, and y3 in the genetic data correspond to the (x,y) coordinate of the three vertices of the triangle. The coordinates are scaled so that all the x-coordinates cover the width of the image evenly and all the y-coordinates scale along the height of the image evenly. The red, green, blue, and alpha values for each triangle determine its color and transparency. An alpha value of 0 is completely transparent and an alpha value of 255 is completely opaque. Since the triangles are rendered in a certain order, a certain amount of overlap will occur which will change the resulting image. As a result, the order of the triangles within the genotype is as important as the actual characteristics of the triangles themselves. A great triangle gene doesn't help at all if it is covered up by another one.

After an individual's solution bitmap image has been rendered, its raw fitness is calculated. The raw fitness is determined by iterating through every pixel of the target image (all x,y) and accumulating the result of the expression $((\text{target}[x][y].\text{red} - \text{solution}[x][y].\text{red})^2 + (\text{target}[x][y].\text{green} - \text{solution}[x][y].\text{green})^2 + (\text{target}[x][y].\text{blue} - \text{solution}[x][y].\text{blue})^2)$ to the total raw fitness count of the individual.

That value is then subtracted from the worst possible value $(255 * 255 * 3 * \text{image_width} * \text{image_height})$ to create a new value, which equates higher fitness with higher values. After the fitness of all individuals in a generation's population has been determined, a fitness scaling operation is performed so that the best individual will always be 10x more likely to get chosen for reproduction than the worst individual. The scaling expression used is:

$$\text{scaled_fitness} = \text{unscaled_raw_fitness} - \text{worst_raw_fitness} + ((\text{best_raw_fitness} - \text{worst_raw_fitness}) / 9)$$

Figure 3. Rendering of an individual's genetic data into an image

Background			Triangle 1 Data										Triangle 2 Data									
R	G	B	A	R	G	B	X1	Y1	X2	Y2	X3	Y3	A	R	G	B	X1	Y1	X2	Y2	X3	Y3
200	255	200	255	230	30	50	40	50	200	80	150	220	127	40	80	250	70	130	220	40	210	210
Individual's Properties:													Rendered Image:									
Background Color Red = 0.78 Green = 1.0 Blue = 1.0 Triangle 1 Alpha = 1.0 (Opaque) Red = 0.90 Green = 0.12 Blue = 0.20 Vertex 1 = (0.16, 0.20) Vertex 2 = (0.78, 0.31) Vertex 3 = (0.59, 0.86) Triangle 2 Alpha = 0.5 Red = 0.16 Green = 0.31 Blue = 0.98 Vertex 1 = (0.27, 0.51) Vertex 2 = (0.86, 0.16) Vertex 3 = (0.82, 0.82)																						

Fitness proportionate selection is done using this scaled fitness. Fitness scaling insures that the best individuals will always be selected more frequently than the worst individuals regardless of the scale or differences between their raw fitness values.

There are five genetic operators available for the GA to use when creating the next generation of a population. These operators are reproduction, crossover, triangle swap, bit mutate, and byte mutate (see figure 4). Reproduction simply copies an individual, unchanged, from one population to the next. The crossover operation selects two individuals and swaps every bit after a randomly selected bit position between the two individuals. The triangle swap operation selects one individual, swaps the 10-byte sequences describing two different triangles in the genetic data, and places the new individual into the next generation. Bit mutate copies an individual to the next generation after flipping one randomly selected bit in the individual's genetic data. Byte mutate copies an individual to the next generation after setting a random byte (byte aligned) in the individual's data to a random value between 0 and 255.

The triangle swap operation is a new operation that is useful for allowing a way for two triangles in the image to swap rendering order. The swapping allows two triangles to trade place in a three dimensional sense as the triangle in the foreground (rendered later) will be swapped more to the background (render earlier) and vice versa. The byte mutate operation is useful for trying out values in byte positions that might be difficult to reach mutating one bit at a time. Local optima can be reached within a byte value that might, in some situations, require more than one bit-flip to escape. The byte mutate operations allows a way around that situation.

The probabilities of using the various operators are shown in table 2. These probabilities were constant for all the runs detailed in the results section. The mutation probabilities were chosen to be relatively high because of the large number of possible schemata in the search space. Mutations are essential in this application to keep the population from becoming too similar and to make up for the large number of schemata that will not be present in the first generation.

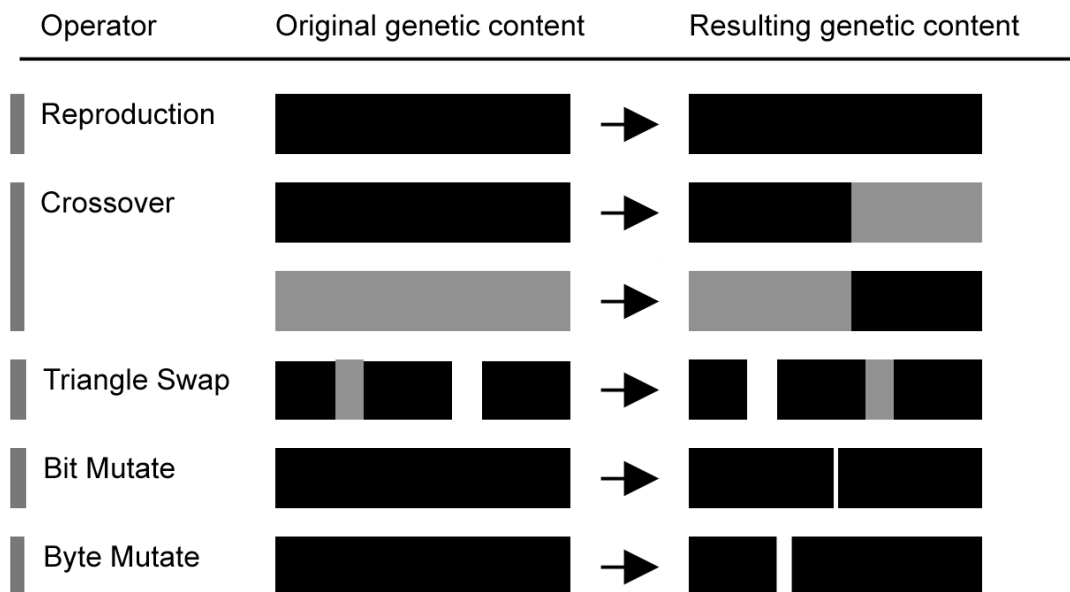
Table 2. Genetic operator probabilities

Reproduction	Crossover	Triangle Swap	Mutate Bit	Mutate Byte
12.5%	50%	12.5%	12.5%	12.5%

In addition to using the five genetic operators described above, an elitism operator is also used to insure that the best individual of any given generation makes it into the next generation. Specifically, the reproduction operator is always used on the best individual of a population to create the first member of the next generation. As a result, the best individual of any generation will always be as good as or better than the best individual of the last generation.

The software implements the genetic algorithm in this manner, continually tracking and saving the best individuals of each Nth generation until it is interrupted or a pre-specified number of generations is run. The fitness of individuals that are reproduced directly into the next generation have their raw fitness copied as well to minimize on the amount of redundant work used to recalculate the fitness of reoccurring individuals. The software was run on a HP Visualize C3000 Workstation running HP-UX 11.0 for the experiments detailed in the results section.

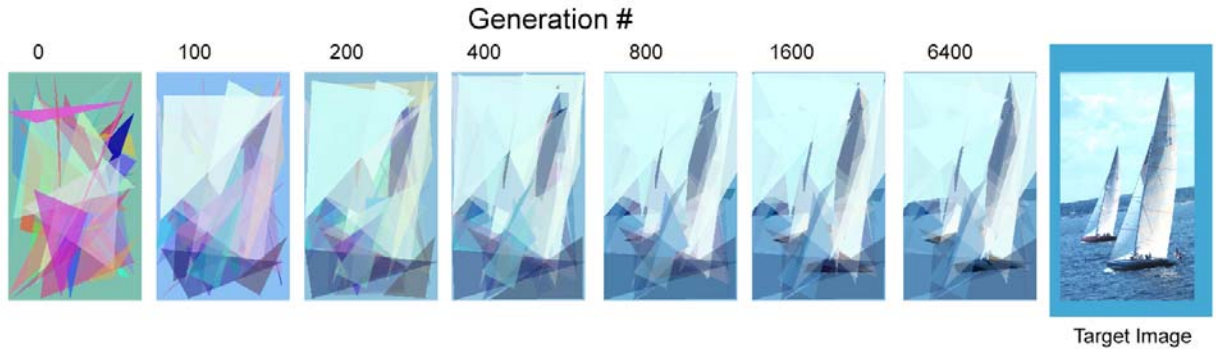
Figure 4. Genetic operators



Results

The software described above was run a number of different times using different target bitmap images and different values for T, the number of triangles to use in the vector-based graphic. In each run, the fitness of the best individual from each generation improved dramatically over the course of the run. The best individuals from the first generations resembled little more than a jumble of random triangles while the individuals from the last generation clearly resemble the original image (see figure 5).

Figure 5. Evolution of a vector-based image



The results of some of the runs are illustrated in figure 6. Table 3 lists the number of triangles used, the number of generations run, and a “percent correct” value for each run. The “percent correct” value is a metric designed to linearly describe how close to perfect each solution was. This percentage is derived by finding the sum of the absolute differences between the red, green, and blue intensities for all corresponding pixels between the solution and target images. A similar value for the theoretically worst possible solution is also found and the “percent correct” value for the solution is calculated as $100\% * ((\text{worst_value} - \text{solution_value}) / \text{worst_value})$. The resulting percentage will be 0% if the solution is the theoretical worst and 100% if it exactly the same as the original. This metric, instead of the raw fitness, was used to measure how good the solution was since it provided a more linear relationship between the best and the worst possible solutions.

Figure 6. Visual results of five runs





Original (Target) Image	Vector-Based (Solution) Image
	
	



Table 3. Results table

Target Image	Percent Correct	Number of Triangles	Generations Run
Union Jack	81.37%	20	20000
Pumpkin	93.24%	50	11000
Stuffed Toy	90.84%	80	10000
Dog	86.04%	60	10000
Sailboats	92.30%	40	17000

Table 4 presents data on how much compression the vector representation of the image receives. The size of the vector-based graphic is $3 + (10 * T)$, as show in figure 2. The size of the original bitmap image is calculated as $3 * \text{width} * \text{height}$ (3 bytes per pixel). A real bitmap file format would also include several more bytes of header information but those have been overlooked for this comparison.

Table 4. Compression results

Original Bitmap Image				Vector-Based Solution Image		
Name	Width	Height	Size (bytes)	Triangles	Size (bytes)	Compression Achieved
Union Jack	116	61	10788	20	203	1.882%
Pumpkin	162	179	86944	50	503	0.579%
Stuffed Toy	170	166	84660	80	803	0.949%
Dog	154	140	64680	60	603	0.932%
Sailboats	82	140	34440	40	403	1.170%

Discussion of Results

In trying to determine the success of the process, it should be noted that in this application, the value of a solution can be measured in a quantitative way but it is the qualitative measure that is the most important. The main goal of the process is to convert a visual representation of something from one format to another. Whether or not the solution actually “looks like” the original image is a subjective quality. A very fit solution that fills the background of a portrait perfectly but omits the facial features of the subject probably would not be considered a good solution even though the numbers say it is.

With that said, cold, hard, objective data on how well the process works does exist and does speak, in a limited manner, as to how well the solution represents the original image. By looking at how closely the pixels from the solution image matched their corresponding pixels from the target image, it appears that the picture of the Union Jack performed considerably worse than the rest of the images. Since the Union Jack image is, itself, just a composition of several red and blue polygons, it would be a trivial task to manually arrange several triangles to match the image almost exactly. The genetic algorithm, however, had a much harder time creating a similar likeness with 20 triangles. Instead of fitting a number of red and blue opaque triangles into place, as a human would do, the GA placed triangles all over the place, many of them with intermediate pink and light blue colors since they covered both red and white or blue and white regions in the target image.

This method of filling in exactly placed triangles with an average of the colors they covered worked much better for the photographic images the GA was run with. All of the major outlines and color regions were well represented in those photograph-based solution images. One thing that was missing in all of the runs, however, was the presence of smaller details in the image such as eyes, leaves, and spots on pillows. One good reason for this is probably the fact that the GA simply wasn’t provided with enough triangles to account for all of the smaller details in an image. The GA’s limited supply of triangles were much better utilized insuring that the colors of larger regions were closer to their actual value. Triangles that filled small details would not increase the fitness as much as triangles that filled broad areas with the correct colors.

The compression numbers achieved by the runs, while impressive, are almost completely arbitrary because there was no threshold quality demanded for a solution image to reach for it to be considered an acceptable compression. The compression values were fixed from the start regardless of how well or poorly the resulting image was. An incredibly detailed image would always achieve the same compression statistics as a very simple image using this process where the number of triangles is fixed. In a more complicated system, where the GA could increase or decrease the number of triangles in an individual based on how detailed the target image was, the compression numbers would be much more meaningful. As they are now, they simply serve to demonstrate how effective a vector-based image of a given size can be at visually representing a corresponding bitmap image of a much larger size.

Conclusion

The experiments in this paper have shown that the genetic algorithm is, in fact, a viable method for generating vector-based graphics from existing bitmap images. Although there are some drawbacks to the method, such as run time and computational resources used, the method is easy to implement and effective. With this application, like so many others, the genetic algorithm provides a straightforward way of finding a solution to a problem you might not understand or have time to figure out otherwise. The quality of the results, while not perfect, can most likely be improved upon by selecting better values for runtime parameters such as operation probabilities, population size, and the number of triangles available for rendering.

The quality of the output of the GA runs also appears to be somewhat dependant on the type of image it is trying to reproduce. Photographs with large fields of color seem to give the best results while images with lots of small sharply contrasting details and those with hard, straight edges don't appear to work as well. Although the methods described in this paper would not likely be a good way of compressing important, detailed photos, or for sending vacation photos over the internet to friends, a better implementation could prove useful for other, less detail oriented applications.

Future Work

There are several directions this work could be taken in the future. One would be to try and improve the method by implementing a variable length GA that could increase or decrease the number of triangles it uses dynamically. Another addition would be to increase the power of the rendering engine so that additional shapes and forms could be represented in the genetic data.

Another future direction would be to implement software that would attempt to create vector-based video where the geometric primitives could extend into time as well as across the two dimensional face of the current video frame. By viewing video as a three dimensional object, the genetic algorithm could be used to find a way of organizing three dimensional objects within the 3 space that would color the planes of different frames in the areas where they intersected.

References

Goldberg, David E. 1989. *Genetic Algorithms in search, optimization, and machine learning*. Addison-Wesley