Programming Languages for Compressing Graphics

Morgan McGuire, Shriram Krishnamurthi and John F. Hughes

Computer Science Department
Brown University
Providence, RI, USA
Contact: sk@cs.brown.edu

Abstract. Images are programs. They are usually simple instructions to a very specialized interpreter that renders them on screen. Image formats therefore correspond to different programming languages, each with distinctive properties of program size and accuracy. Image-processing languages render large images from small pieces of code. We present Evolver, a language and toolkit that perform the reverse transformation.
The toolkit accepts images in conventional graphics formats like JPEG and uses genetic algorithms to grow a program in the Evolver language that generates a similar image. Because the program it produces is often significantly smaller than the input image, Evolver can be used as a compression tool.
The language balances the tradeoff between having many features, which improves compression, and fewer features, which improves searching. In addition, by being programmatic descriptions, the rendered images scale much better to multiple resolutions than fixed-size images. We have implemented this system and present examples of its use.

1 Introduction

The growing volume of data on the Web has resulted in enormous demands on network bandwidth. Improvements in network access, such as the increasing availability of high-speed connections outside commercial venues, have simply resulted in correspondingly larger data and the greater use of bandwidth-hungry formats. Web designers have come to rely on such availability, producing denser sites, and users in turn have higher expectations. As a result, the principal cost for many sites is simply that of leasing sufficient bandwidth. This cost dwarfs that of providing and maintaining content. Since visual formats are central to many Web interfaces, their compression is of key importance in making existing bandwidth more effective. Keeping visual formats small will have obvious benefits to Web designers, site administrators and users.

The common approach to compressing data is to use a standard lossless compression algorithm, such as the many variants of Lempel-Ziv [19], which reduce redundancy using a dictionary. Some formats like JPEG [18] are tuned specifically for graphics by accepting imperfect (lossy) compression and reverse

engineering the human visual system to determine where compression artifacts will have the least perceptual impact. There are unfortunately limitations with these approaches that preclude many improvements:

- They still result in fairly large files. The benefits of compression are limited by the artifacts introduced when the nature of the image does not match the compression technique. For instance, JPEG was designed for images whose frequency decomposition to is similar to photographs. It is less effective on images with many high-frequency components, e.g., diagrams with sharp distinctions between differently colored regions or images containing text. GIF was designed for images just like these, but produces dithered and poorly colored results for natural images like photographs. Neither performs well when both kinds of data are contained in a single image.
- The resulting images are at a fixed resolution. Proper display on devices of varying resolution will only grow in importance as new devices such as PDAs are deployed as browsers.
- Because the output format is fixed, the server administrator cannot trade space for time. In particular, running the compression utility, which is a one-shot activity that benefits all users, for a longer time will not result in a smaller or better image.

Designers address some of these problems by replacing static entities with programs that generate them (e.g., Flash, SVG objects). Even a basic image is, in the abstract, a program. For example, a GIF file contains a series of run-length encoded rasters. The run-length blocks can be considered instructions for a very primitive machine which produces its output in the form of an image.

Formats make the idea of transmitting a program as a form of compression more explicit. Formats such as Flash, Shockwave and SVG [3] use a richer underlying language that keeps objects distinct and resolution-independent, allowing implementors to describe vector-based graphics. Flash and Shockwave also add time-varying instructions and loops to enable animation. The virtual machines for these programs are Web browser plug-ins. Using Java to generate animations is an extreme example of this same principle, namely *programmatic compression*:

> a program is often much smaller than the output it generates.[1]

A program is also much more capable of scaling by resolution (recall Knuth's METAFONT [11] for generating fonts from typeface descriptions).

The Flash approach reflects a valuable principle, but has many practical shortcomings. It requires the manual construction of programs from constituent arcs, lines, shades, and so forth. Many complex entities, such as texture-mapped regions, are difficult or impossible to specify using so-called vector graphics primitives. Professional Web designers thus prefer to use tools such as Adobe Photoshop to generate high-quality images with effects like drop shadow, texture, and compositing of scanned/digital photographs. An artist can create such an

[1] For a non-visual example, consider programs that generate the digits in the decimal expansion of π.

image in Photoshop and import it into a tool like Flash as a JPEG, but this immediately loses the benefits of a programmatic description, such as effective scaling and extreme compression. The result is a comparably large and pixellated image embedded in an otherwise small and resolution-independent vector description. Ideally, designers should generate images with their favorite general purpose tools like Photoshop; *the compression technique must integrate into their existing design flow.*

The rest of this paper is organized as follows. Section 2 outlines the principles underlying Evolver, our language for graphical image compression. Section 3 provides the details of Evolver's language and tools. Section 4 describes some of our results. Section 5 discusses related work. Finally, section 6 offers concluding remarks and discusses directions for future work.

2 Generating Image Descriptions

The display software for a GIF must decode the format's run-length encoding, thereby effectively acting as an interpreter. Similarly, JPEG images depend on the encoding of small blocks of an image; the display software must again decode the data for display. As a more extreme example, various "fractal compression" algorithms rely on more complex encodings; iterated function systems [1, 4] and partitioned iterated function systems [10] rely on the idea that small parts of an image may resemble translated, rotated, scaled-down copies of other larger portions; a simple version of the key idea is that some savings can be found by representing the smaller portion via this transformation rather than directly.[2] Each of these compression algorithms can be seen as exploiting some known characteristic of images: the tendency to be scan-line coherent, the tendency to have large blocks with small variation, or the tendency to have little pieces that look like scaled versions of larger pieces. The result of each of these insights is a particular program (a decoder) that takes a certain amount of data (the "GIF" or "JPEG" or "PIFS" image) and converts it to an image (a rectangular array of colors).

Note that certain types of image are well-suited to each of these approaches: scanned-text compresses pretty well with GIF, in part because of the long "white" runs between the lines. Photos of natural phenomena seem to be well-suited to iterated-function-system compression. And JPEG works well on interior photos, where things like walls present relatively large, relatively constant areas. These techniques tend to produce large files or introduce artifacts when applied to the wrong kind of images.

One might therefore imagine an image format that stored an image either as a GIF or a JPEG or a PIFS, by trying each one and selecting the one with the best compression results; the particular type used could be indicated with a few bits in a header. But from this, a more general approach suggests itself: why not consider *all* possible programs, not just the handful that have been written down

[2] Discovering the correct transformations is the hard part of the compression process.

as JPEG or GIF decompressors, that can generate an image and simply record the one that best generates *this* image? (Note that in this case, the decoder and the data it operates on are all considered part of "the program"; indeed, the decoder-and-data model of generating images is only one of many possibilities.)

While the idea of searching "all possible programs" seems ludicrous at first blush (there are an infinite number of them, for example), searching a rich collection of programs turns out to be feasible with some insight into the design of the language in which the programs are written and the goal of the program.

Evolver

We present Evolver, a collection of tools for visual image compression. Because artists are skilled at the use of image manipulation software, not programming, Evolver presents a simple interface: the designer provides an image (the *source image*) built with their favorite tools. Evolver reduces this image to an Evolver language program (the *description*) which, by rendering a reasonable approximation of the image and being small, exploits the power of programmatic compression.

Evolver has two parts. The first program is an encoder, which consumes the source image and produces its description. It does this by searching the space of all legal descriptions to find ones that generate an image that is visually similar to the source image. The second, a decoder, is a fast interpreter for the Evolver language, which renders the description as an image for the client. The decoder is currently implemented in Java [7], and is hence easy to deploy in Web contexts.[3]

Evolver confronts two hard problems. First, it is difficult to search the space of all programs, because the space is large, and small changes to the source of a program can lead to large changes in its output. Second, it's difficult to create an objective measure of visual similarity. Evolver therefore employs randomized search algorithms. Our current implementation uses *genetic algorithms*, following Karl Sims's ideas from a decade ago [17]; that is, *Evolver breeds a program that generates the desired image*. We could use other algorithms such as simulated annealing instead.

Because it uses an optimizer to search, the effectiveness of Evolver depends on the choice of an appropriate fitness function. We could obtain a perfect image match by subtracting the current approximation from the original and requiring that the difference be zero; this would, however, take unreasonably long to converge. This also fails to exploit the feature that several different descriptions can render identical-looking images within the tolerance of the human visual system. We instead use more flexible metrics that account for desirable visual properties, such as matching texture and shape, rather than individual pixel values. For example, it is acceptable to compress one white noise image into another

[3] The Java bytecode for the decoder is substantially larger than the descriptions it decompresses into images; we assume that for practical deployment the Java decoder would be downloaded once and used many times, so its size does not pose a problem.

where no individual pixel matches, so long as we preserve the statistical properties of the noise. We are not aware of a pre-existing compression technique that uses the property that two noise signals with the same statistics are visually indistinguishable.

Evolver rewards longer runs by producing higher quality matches. Some of these programs grow in size to improve the approximation. Over time, however, Evolver may also find smaller programs that generate the same image. Therefore, Evolver presents the designer a range of images corresponding to the original; the designer can pick the smallest or best, or even choose several for use in different contexts.[4] The image's description (an Evolver program) then becomes the object distributed to clients.

3 The Evolver Language

Evolver's success relies crucially on the choice of language. Not all choices of languages automatically accomplish compression. For instance, consider the trivial, single-instruction language whose one instruction consumes the coordinates and color of a point and shades it accordingly. Slavishly translating each pixel in the source image into an instruction in this language would accomplish little, and might well expand the size of the description. In contrast, because many images contain considerable redundancy, adding abstraction mechanisms to the language can lead to programs that account for this redundancy, thus removing it in the image's description. Run-length encoding is a simple example of this: a "repeat n times" mechanism in the trivial language above would allow it to implement run-length encoding.

The benefit of a larger language is obvious: the more powerful its instructions, the greater the likelihood that a concise description exists in the language. Another benefit of a larger language is that it allows us to encode something of our knowledge of the structure of images. The `Collage` primitive of Evolver, described later in this section, is a good example of this kind of domain-specific construct.

The benefits that ensue from large languages mask a significant shortcoming. The larger the language, and the greater its parameterizations, the harder the genetic algorithm needs to work to find even valid programs,[5] much less ones that render a useful approximation to the source image. *This is the central tension in the design of Evolver's language.* To understand the explosion of search

[4] The smallest and best criteria can be combined into a single criterion by accumulating the per-pixel color value differences between Evolver's result image and the source image and counting the number of bits needed to encode those deltas against each description. This is overly strict because different pixel values do not necessarily produce visually different images. In practice, there is no established algorithm for reliably measuring the perceptual error in an image approximation and we are left with appealing to a human observer to obtain the best results.

[5] Sims finessed this issue by creating a language/breeding system in which every possible mutation was a valid program. We follow a similar approach, and discuss this issue further in section 5.

complexity that results from adding language features, consider variable binding, a feature that is useful in programming languages used by humans but is particularly difficult for a search program to use effectively. The search program must both bind a variable and use that binding for it to contribute compression. Evolving both of those code fragments simultaneously in a way that increases match quality may be a low probability occurrence. In contrast, highly nested and self-similar programs are easy for the search program to generate but look nothing like the programs a human would write. Furthermore, Evolver optimizes globally over an image, and may perform filter steps that affect different portions of an image differently, in clever ways a human is unlikely to engineer.

The Evolver language is a simple, functional language, based on that used by Sims. It provides a rich set of primitives that render various shapes on screen. We describe these primitives in groups, with some brief rationale given for each. All primitives operate on (and produce) *matrices*, which correspond to pixel samples in an image; thus a matrix is a 3D array of values arranged into a rectangular array of color *vectors*, where a vector is a triple of *scalars*, and a scalar is a number between -1.0 and 1.0. In some cases, it's natural to do something to the *first* entry of each triple; the array of all these "first entries" will be called the "first channel" of the matrix. Because at display time, each triple is interpreted as the red, green, and blue components of a pixel, we'll sometimes speak of the "red channel" as well.[6]

The interpretation of an Evolver program depends on the *dimensions* of the final image; all matrices in the program are of these same dimensions, which enables resolution independence. Some of the primitives in the Evolver language use stock images as constants, which do not mesh well with resolution independence. To address this, a high-resolution stock image is appropriately filtered to the matrix size and then sampled at the given size. This means an Evolver program is really parameterized by the output image dimensions; the primitives are designed to have the characteristic that a low-resolution result of a program looks like a scaled-down version of a high-resolution result of the same program.

We allow values to be coerced between scalar, vector, and matrix types. Vector values can be derived from a matrix by operations like "compute the average color the image." This has the advantage that it is reasonably scale-independent: whether the images we're working with are 10x10 or 300x300, the average of all pixels is roughly the same. Scalars can be obtained from vectors by chosing a single channel or the average. A scalar is coerced to a vector by repeating it three times; a vector is coerced to a matrix by repeating it the necessary number of times.

The full Evolver language grammar in BNF notation is given by:

```
ELValue := ELMatrix | ELScalar | ELVector
ELOperator := Add | Sub | Blur | Noise | ...
ELCall := ELOperator x ELExpression*
```

[6] The mapping from the interval $[-1.0, 1.0]$ to colors is due to Sims: negative values are clamped to no intensity; numbers between 0 and 1 map to the full range of intensities.

```
ELExpression := ELCall | ELScalar | ELVector
ELMatrix := ELVector*
ELVector := ELScalar x ELScalar x ELScalar
```

where an `ELScalar` is a real number between -1 and 1.

3.1 Fitness Function

Our fitness function, which may be simplistic by the standards of the computer vision community, attempts to match properties of images that are known to affect the human visual system. The properties we use are:

- edges,
- per-pixel intensity,
- pixel color,
- average color over large regions (image segments),
- average color over small regions.

We compute these properties of the input and current image, and sum the differences along each of these metrics. We choose the sum to avoid preferring generations that succeed in a few metrics but greatly fail in the others.

Edges are a crude measure of frequency data, so an area in an image that has high frequency will appear densely populated with edges. High-frequency areas (such as images of leaves and grass) will match poorly with low-frequency images such as gradients and constants, and well with other high-frequency areas such as noise and stock images. We tried to employ the magnitude of the gradient of the image, which is a better measure of local frequency content, but found that it gave poor matches compared with the edge detector.

3.2 Language Primitives

The language grammar reveals that the core of Evolver lies in its set of primitives. The details are not essential to the larger point and are too large to print here. Instead, we provide detailed information on the Web:

`http://www.cs.brown.edu/people/morgan/evolver/`

Here, we describe the categories of primitives with some commentary on the graphical motivation for each type of primitive.

Variation Sources To produce compact representations of nearly-constant areas of an image (which are often good targets for compression), it helps to have some functions that produce slowly-varying signals, which can then be arithmetically mapped to generate periodic signals (via `sin` or `cos`), or can be used directly to represent slowly-varying regions of the input images. Similarly, band-limited noise has been known to be very valuable in texture generation [5]; we therefore include primitives that generate this directly. In all cases, we use Perlin's noise

Fig. 1. A few of Evolver's image primitives.

function, but with a constant "seed" so that the results are reproducible. Finally, purely statistical information about an image can help to generate patterns that may be likely to resemble those appearing in other images "like" this one; we therefore have primitives that are images such as those shown in figure 1. (Note that none of these images was actually in the test data set.)

Mathematical Transformations Following Sims, we allow basic arithmetic operations on matrices, and element-wise transformations such as mathematical (sine, cosine) and logical (and, or) operations. Some specialized operations include `ExpandRange`, which expands the dynamic range of a matrix's contents from (0, 1) to (-1, 1). This has the effect of enhancing the contrast in an image, while reducing the average brightness.

Color Some primitives reinterpret or convert channels between RGB and HSV.

Geometric Pattern Generation Tools A few primitives help with the representation of images that have some geometric regularity to them, by providing various natural transformations. These include symmetric mirrors, splits and zooms.

Combinations Naturally, some of the most important primitives are those that combine images in interesting ways; ordinary arithmetic operations have been discussed above, but the following operations are richer, and preliminary evidence suggests that `Rotate`, `HueShift`, and `Interpolate` are powerful enough to be selected often by Evolver.
`Rotate(matrix, matrix)`: Rotates the first image by the average value of the red channel of the second argument (averaged over whole image); tiles missing areas.
`Interpolate(matrix, matrix, matrix)`: Linearly interpolates the first and third arguments using the second as the blending value.
`Distort(matrix, matrix)`: Displaces the pixels of the first argument by the values of the second (as if refracting through glass).
`EnvironmentMap(matrix, matrix)`: Interprets the first argument as a height map and uses the second as a color environment map (as if reflecting the second

image).
`HueShift(matrix, matrix)`: Shift the overall hue of the first argument by average red value of the second argument.
`Collage(matrix*)`: Colors different segments with different textures.

The last of these deserves special comment: because it is hard for Evolver to determine how to partition the image, we start the search with a "Collage" primitive. This depends on a first step: we take the target image and compute a "segmentation" of it, breaking it into regions of similar hue. The segmentation algorithm is relatively naive, and tends to produce regions which may not represent terribly precise segmentations, but which are extremely amenable to run-length encoding, and hence can be compactly represented. This segmentation is provided as "data" to the evolver program (and its size is taken into account when we measure compression!); the primitive `collage(m1, m2, ...)` creates an image where pixels from segment 1 are taken from the matrix `m1`, those from segment 2 are taken from matrix `m2`, and so on.

3.3 Linguistic Observations

Many of the primitives were chosen because they worked well in Sims's original work; others were included because they have proven useful in image processing. Evolver's language, as with most other programming languages, grows through an evolutionary process. The language we currently have reflects a very capable set of primitives that renders a diverse stable of images.

Our primitive set is not minimal by any means: several primitives can be built from simple combinations of other primitives. They were included as a way of injecting domain knowledge into the search process to make it converge faster. For example, a translation operation can be generated as an application of the form `distort(matrix m, matrix c)` where `c` is a constant matrix, and indeed, Evolver often uses this idiom; thus including `Translate` as a new primitive is natural.

In general, we inject domain knowledge at three points:

- The selection of primitives, as explained above.
- The mutation strategy: one of our mutations involves adjusting a scalar by a small amount, for example. Large adjustments caused too-drastic changes in the results; tiny adjustments made it hard to ever "get where we needed to be." Hence we chose a modest average adjustment to scalars. Similarly, in the mutations that wrap an expression in a larger one or trim away part of an expression, we recognized the need to balance the frequencies of these operations so that the size of expressions did not tend to grow without bound.
- The fitness function: by adjusting the fitness function to be sensitive to approximate edge-placement, we made substantial improvements over an earlier L^2-distance function. The better that one can match the fitness function to the human visual system, the better the results will be.

4 Experimental Results

Our test data set was a series of outdoor photographs by Philip Greenspun available on `photo.net`. We loaded each image into the encoder and allowed it to run for a fixed period of time. In each case, the encoder segmented the image and ran for 10 minutes on each segment, then combined the segments using the `collage` operator and ran for an additional 10 minutes on the combined expression. The parameters used to tune the genetic algorithm will be available on our website.

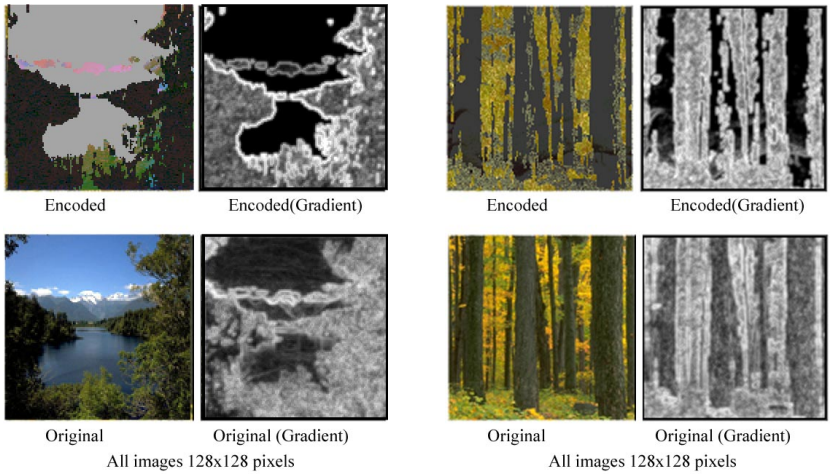


Fig. 2. Some uses of Evolver.

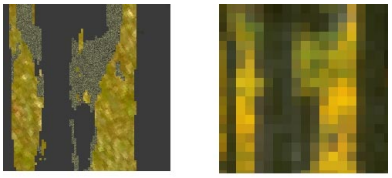
The figure on the left shows a poor result from Evolver. The source image is in the lower left corner. This is an image of a lake surrounded by trees, with mountains in the distance. In the lower right is an image showing the magnitude of the gradient of the source image. This is not used by the encoder but is useful for evaluating the quality of the results and understanding the matcher. Light values in the gradient correspond to areas of high frequency, like leaves, and dark values correspond to low frequencies like sky.

The upper left image shows the image after compression by Evolver and the upper right shows the gradient of this image. While the encoding contains visible artifacts and is thus inferior to the compression available through a GIF or JPEG, it demonstrates that the evolutionary approach is clearly feasible. Evolver correctly matches the hue and intensity in the image, creating bright green areas where the trees were illuminated and dark green areas where the trees were in shadow. The sky and lake are a uniform grey, averaging the color of the lake and mountains. The clouds are pink. Looking at the gradient, we see that the encoder matched high frequency textures to the tree areas and

low frequencies to the lake and sky. The mountains are absent entirely. This is because the segmentation failed to distinguish between the sky and mountains, so Evolver created a single texture for both of them.

The figure on the right shows a much better result. The source image is a close-up view of some maple trees. After 40 minutes, Evolver successfully matched both the frequency, color and texture of the source image. The encoded image has minimal artifacts, and thus indicates that Evolver can successfully function as a compression algorithm.

As a multiresolution experiment, we compressed an 128x128 original image using Evolver and as a 64x64 JPEG. The goal was to store the image using very few bits without losing all detail. We then decompressed and displayed the image at 768x768. Evolver's procedural representation preserves high frequency detail. The small JPEG image is blocky when viewed at 12x the encoding size. Below are zoomed in views of a few pixels from each to show the difference.



Evolver JPEG

Virtual environments often suffer from low texture resolution. When the viewer is very close to a surface, textures appear either blurry or blocky. It has long been known that procedural textures are a solution to this problem. Traditionally, procedural textures could only be used for textures like wood grain and checkerboards that were easy for humans to synthesize algorithms for. Evolver is important for this problem because it is a method for taking an arbitrary image texture and making a procedural texture from it.

Note that Evolver will add texture at higher frequencies than appear in the original image. This is only mathematically possible because it may add the wrong information; it is not sampling the original scene at a higher frequency but trying to simulate what the results of that process would be. Because Evolver uses natural images as part of its input data set, it is predisposed to creating detail that has the same statistical properties as that found in real images. If Evolver matches a stock leaf texture to a forest, it is likely that zooming in will show individual leaves and not blocks. Of course, it is also possible that from a distance carpet and leaves look the same and on zooming in the viewer will discover that Evolver carpeted the forest instead. The user can control this process by picking an appropriate set of primitive images.

5 Other Related Work

There is a long history of procedural graphics languages. Papert's LOGO language [15], Henderson's Picture Language [9], Knuth's TEXand Metafont [11] are some early and familiar examples. Programmable shading languages like Renderman [8, 16] continue to play an important role in graphics.

Perlin, and Worley independently worked on many procedural texturing algorithms which are summarized in their book [5]. This work all assumes a human writes the code manually, which is prohibitively difficult for complicated textures such as actual photographs.

Barnsley [1] introduced the idea of iterated function systems ("fractals") for compression. These are much richer graphics languages than JPEG or GIF but less general than Evolver, which does not rely on self-similarity for compression.

Massalin's *superoptimizer* [12] tries all sequences of assembly language instructions to find the smallest one equivalent to a given input function. That is, the superoptimizer conducts a comprehensive search of the state space of programs. It therefore clearly represents an extreme instance of optimization. While the superoptimizer finds a globally optimal solution, it is clearly infeasible in the large space that Evolver searches.

Nicholas et al. [13] have studied the problem of *typed* genetic programming to improve the convergence speed of genetic programs. These works are not immediately applicable to ours, because they consider much weaker languages whose type domains we cannot adopt. We believe it is important to use a typed language in Evolver, and intend to do this as future work.

Beretta, et al. [2] describe a technique for compressing images using genetic algorithms. Nordin, et al. [14] describes a similar program for images and sound. Both of these systems use primitive languages (at the machine instruction level) and operate on 8x8 or 16x16 blocks. We build on their results by using genetic algorithms for compression and the observation that dividing the image into separate regions speeds convergence. Evolver differs in that it uses an image segmentation based on objects, not arbitrary blocks, and features a rich image processing language. Our approach avoids the blocky artifacts from these systems and allows Evolver to capture details in a multi-resolution fashion. It also gives Evolver the potential for much higher compression ratios, since 8x8 blocks can achieve at most a factor of 64:1 compression.

Our work is directly inspired by Karl Sims [17], who used genetic algorithms to evolve images on a connection machine with a high level language. In his experiments, the fitness function was the user's aesthetic preference, and the human and computer interacted to form visually pleasing abstract images. We use a language and genetic algorithm similar to Sims but apply it to image compression to synthesize programs that create actual photographs.

6 Conclusions and Future Work

We have presented the Evolver framework, a collection of tools for compressing graphical images. At its heart, Evolver consists of a simple, functional graphics

description language, and its corresponding interpreter, which resides on client computers. Given an image to compress, Evolver breeds a description that generates an image that the designer considers acceptably close to the source image. This program is generated once on the server, and used many times on numerous clients.

Because the image is rendered by a program, Evolver offers many benefits not found in traditional formats and compressors. Image descriptions are resolution-independent, so the same description can render both a thumbnail image and the actual one, with graceful degradation. The genetic algorithms reward patience: by running longer, they can produce better approximations to the original image, or find smaller programs that generate acceptable approximations. The resulting image description can be extremely small, and can be replaced with smaller descriptions as they become available. Different genetic algorithms are better-suited to different kinds of images, so as Evolver encounters new families of images, it simply needs new mating techniques; since the output is still a rendering program in the same language, the client does not need to make changes. Even if the language does grow, an enriched interpreter would still handle any older images.

So far, we have only discussed the use of Evolver to compress static images. We are also conducting experiments on compressing animations. On the one hand, animations appear to offer a considerable challenge. It can be fairly time-consuming to generate a description of even a single static image; how much longer would it take to generate a small program that generates a *sequence* of images? Indeed, this problem seems practically intractable.

There are, in fact, several feasible ways of approaching animations. One obvious approach is to treat each frame individually, apply Evolver to each frame in turn, and add a single instruction to its language to encapsulate a sequence of frames. We can improve the convergence by exploiting temporal locality by using the description for one frame as the initial description for the next.

In the ideal case, Evolver will incorporate a second language, dedicated to animations. This second language would capture attributes such as a consistent darkening of all pixels in a region (perhaps indicating nightfall). This again exploits the programmatic principle that a large difference in bit-wise information may be captured by a small set of instructions. Various MPEG encoding standards use "difference" data of this sort, which we hope to exploit for designing the animation language.

More importantly, we see Evolver pointing to numerous research opportunities for the programming languages community. First, the genetic algorithms in Evolver need a type system to guide the generation of programs; currently, Evolver uses fairly ad hoc techniques to prevent the generation of invalid programs. Second, there are likely to be several static analyses that, with small extensions to the intermediate language, will encourage "suitable" programs (at the expense of some diversity), yielding quicker convergence. Third, Evolver may be able to benefit from some of the features of existing graphical languages, such as FRAN [6]; we were forced to use Java because Haskell currently lacks the rich

graphics library and widespread applet support necessary to deploy Evolver, but the language's design in no way precludes, and indeed encourages, implementation in a functional language. Finally, Evolver points to a new criterion for the design of some domain-specific languages: to be suitable for creation of programs by other programs, especially through a simple but highly iterative process such as evolution.

Ultimately, our goal is not to develop radical new compression schemes—we defer to others for whom that is a primary research interest. Instead, we believe that Evolver's true value lies in the philosophy that it embodies:

- Many problems in science and engineering are solved by changing representations. Programming languages are pervasive, and can therefore serve as a useful alternate representation in many domains. Converting a domain's data into code makes it possible to leverage the techniques and tools of programming languages—such as semantics-preserving program transformations and interpreters, respectively—to tackle difficult problems.
- Leveraging steerable, probabilistic search techniques, such as genetic algorithms, permits the selective injection of domain knowledge into optimization problems.
- The burgeoning availability of cycles creates possibilities for whole new styles of programs. These may employ, and can even exploit, non-standard notions of "correctness". In Evolver, for instance, the validity of an approximation is determined entirely by the judgment of the designer's visual sensory system.

We believe that this is an important design pattern that can be used to effectively open a radical new approach to hard problems in several domains.

Acknowledgments

This work has been supported in part by the NSF Science and Technology Center for Computer Graphics and Scientific Visualization, Adobe Systems, Alias/Wavefront, Department of Energy, IBM, Intel, Microsoft, Sun Microsystems, and TACO, and NSF Grant ESI 0010064. Nature photographs courtesy of and copyright to Philip Greenspun (`http://photo.net/philg/`). Fish and sunrise images courtesy of and copyright to Morgan McGuire. We thank Nick Beaudrot for his help writing the toolkit.

References

1. M. F. Barnsley and A. E. Jacquin. Application of recurrent iterated function systems to images. In *Proceedings SPIE Visual Communications and Image Processing '88*, volume 1001, pages 122–131, 1988.
2. M. Beretta and A. Tettamanzi. An evolutionary approach to fuzzy image compression. In *Proceedings of the Italian Workshop on Fuzzy Logic (WILF 95)*, pages 49–57, Naples, Italy, 1996. World Scientific.
3. World Wide Web Consortium. Scalable vector graphics (SVG) 1.0 specification, 2001. http://www.w3.org/TR/SVG/.

4. S. Demko, L. Hodges, and B. Naylor. Construction of fractal objects with iterated function systems. In B. A. Barsky, editor, *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, volume 19 (3), pages 271–278, San Francisco, California, July 1985.
5. D. Ebert, K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach, second edition*. AP Professional, 1998.
6. Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.
7. James Gosling, Bill Joy, and Guy Lewis Steele, Jr. *The Java Language Specification*. Addison-Wesley, 1996.
8. Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, volume 24 (4), pages 289–298, Dallas, Texas, August 1990. ISBN 0-201-50933-4.
9. Peter Henderson. Functional geometry. In *Symposium on Lisp and Functional Programming*, pages 179 – 187, New York, 1982. ACM Press.
10. Hau-Lai Ho and Wai-Kuen Cham. Attractor image coding using lapped partitioned iterated function systems. In *Proceedings ICASSP-97 (IEEE International Conference on Acoustics, Speech and Signal Processing)*, volume 4, pages 2917–2920, Munich, Germany, 1997.
11. D. Knuth. *TEX and METAFONT : new directions in typesetting*. Digital Press and the American Mathematical Society, 1979.
12. H. Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 22, pages 122–127, New York, NY, 1987. ACM Press.
13. Nicholas F. McPhee and Riccardo Poli. A schema theory analysis of the evolution of size in genetic programming with linear representations. In *Genetic Programming, Proceedings of EuroGP 2001*, LNCS, Milan, 2001. Springer-Verlag.
14. Peter Nordin and Wolfgang Banzhaf. Programmatic compression of images and sound. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 345–350, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
15. S. A. Papert. Teaching children thinking. Technical Report A. I. MEMO 247 and Logo Memo 2, AI Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1971.
16. Pixar. The renderman interface, version 3.1, 1989. http://www.pixar.com/renderman/developers_corner/rispec/.
17. K. Sims. Interactive evolution of equations for procedural models. In *Proceedings of IMAGINA conference, Monte Carlo, January 29-31, 1992*, 1992.
18. Gregory K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.
19. Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.