Emergent Design and Image Processing: 
A Case Study

Richard W. DeVaul
B.E.D, Environmental Design (1994)
Texas A&M University

Submitted to the Program in Media Arts and Sciences, 
School of Architecture and Planning, in Partial 
Fulfillment of the Requirements for the Degree of 
Master of Science in Media Arts and Sciences 
at the 
Massachusetts Institute of Technology 
June 1999

©1999 Massachusetts Institute of Technology 
All rights reserved
Emergent Design and Image Processing:  
A Case Study

by

Richard W. DeVaul

Submitted to the Program in Media Arts and Sciences,  
School of Architecture and Planning, on May 7, 1999  
in Partial Fulfillment of the Requirements for the Degree of  
Master of Science in Media Arts and Sciences

Abstract

The digital revolution which has changed so many other aspects of modern life  
has yet to profoundly affect the working process of visual artists and designers. High-quality digital design tools exist, but they provide the user with an improved traditional design process, not a radically new way of designing. Conventional digital design tools are useful, but when design software emulates a paintbrush or photo-studio many powerful possibilities of the computational medium are overlooked.

This thesis explores emergent design, a design methodology based on a new process, enhanced interactive genetic programming. The emergent design methodology and tools allow designers to effectively create procedural design solutions (design solutions that take the form of a procedure or program) in a way that requires little or no programming on the part of the designer. The use of preliminary fitness functions in the interactive genetic programming process allows the designer to specify heuristics to guide the search and manage the complexity of the interactive genetic programming task.

This document is structured in the form of a case study, in which the enhanced genetic programming process and emergent design methodology are described through their application to the specific problem of developing procedural image filters for still and moving images. Two interactive genetic programming systems for image filter evolution are described, GPI and evolution++, along with the Sol programming language that was used to create them.

Results from the implementation and use of GPI and evolution++ are presented, including a number of filtered images and image sequences. These results suggest that fitness-agent enhanced interactive genetic programming and the emergent design methodology may play a useful role in the visual design process, allowing designers to explore a wider range of options with greater ease than is possible through a traditional, procedural, or conventional genetic programming design process.

Thesis Supervisor: John Maeda  
Title: Assistant Professor of Media Arts and Sciences
Emergent Design and Image Processing

Thesis Committee

Thesis Advisor

John Maeda
Assistant Professor of Media Arts and Sciences
MIT Media Laboratory

Reader

Mitchel Resnick
Associate Professor of Media Arts and Sciences
MIT Media Laboratory

Reader

Henry Lieberman
Research Scientist
MIT Media Laboratory
Contents

1 introduction 11

2 research overview 13
  2.1 interactive genetic programming . . . . . . . . . . . . . . . . . . . 15
    2.1.1 Karl Sims . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
  2.2 enhanced interactive genetic programming . . . . . . . . . . . . . 16
  2.3 emergent design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3 emergent design — a hypothetical example 19
  3.1 the Cassero Bella Center . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
  3.2 the technical challenges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  3.3 the emergent design process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
    3.3.1 a sophisticated emergent design system . . . . . . . . . . . . . . . 21
    3.3.2 real-time video filtering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

4 creating images 25
  4.1 digital image manipulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
    4.1.1 history . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
    4.1.2 present technology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

5 image filtering 28
  5.1 discrete geometric transforms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
  5.2 a simple procedural image processing framework . . . . . . . . . . . . . . 29
  5.3 discrete data and signal reconstruction . . . . . . . . . . . . . . . . . . . . . . . 31
    5.3.1 continuous coordinates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
    5.3.2 resampling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
    5.3.3 aliasing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
  5.4 color . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
    5.4.1 the RGB color space . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
    5.4.2 the HSV color space . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
    5.4.3 color image filtering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
    5.4.4 the choice of HSV over RGB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
    5.4.5 the complexity of HSV vs. RGB filtering . . . . . . . . . . . . . . . . . 35
  5.5 filter representation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

6 genetic programming for image processing 40
  6.1 automatic programming . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
  6.2 genetic search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

7
6.2.1 genetic algorithms .......................... 42
6.2.2 genetic programming ......................... 44
6.2.3 valid expressions and a robust representation 45
6.3 evolving image filters ............................. 46
6.3.1 designer as fitness function ................. 46
6.3.2 the genetic programming design process .... 47

7 enhanced interactive genetic programming 49
7.1 the APFA framework .............................. 49
7.2 improving the search ............................. 50
7.2.1 computational images ......................... 50
7.2.2 the APFA selection function chain .......... 51
7.3 the extended APFA framework ................. 53
7.3.1 advantages and disadvantages .............. 54
7.4 emergent design ................................. 55

8 emergent design 56
8.1 design theory .................................. 56
8.2 design tools and process ....................... 56
8.2.1 non-procedural design ....................... 57
8.2.2 procedural design ............................ 57
8.2.3 participatory procedural design ............ 58
8.2.4 emergent design .............................. 58
8.3 emergent design and the model of dynamic design 59
8.3.1 justifying the emergent design processes ... 59

9 implementation 61
9.1 a procedural design language ................... 61
9.1.1 the features of an ideal language .......... 61
9.1.2 existing languages ......................... 62
9.2 Sol .............................................. 64
9.3 genetic programming in Sol ..................... 65
9.3.1 operators in Sol ............................. 66
9.4 GPI and evolution++ ............................ 67
9.4.1 GPI .......................................... 67
9.4.2 GPI and APFA ............................... 69
9.5 evolution++ ...................................... 69
9.5.1 interfacing Sol and C++ .................... 69
9.5.2 translating Sol filters into C++ ........... 70
9.5.3 run-time compilation and dynamic linking . 73
1 introduction

Since Ian Sutherland developed the Sketchpad system in the early 60s[28], the field of computer graphics has undergone immense transformation and development. Graphics hardware that cost millions of dollars and filled entire rooms thirty years ago has been replaced by the inexpensive desktop, laptop or even pen-top hardware of the 90s. During my lifetime, I have seen graphics computing go from an esoteric industrial application to a commodity.

The initial applications of graphics computing were restricted to research, industrial or military applications where its high cost could be justified in academic, economic, or strategic terms. The users of these early systems were often the researchers or engineers who created them. As the cost of the hardware decreased, the diversity of the applications and users increased. The advent of the personal computer in late 1970s revolutionized the industry and made access to computer graphics hardware widely available.

From the very beginning, computer graphics borrowed interaction techniques from pre-digital media, such as the drawing metaphor of the Sketchpad system. With the invention of the Graphical User Interface (GUI) at the Xerox Palo Alto Research Center in the 1970s and popularized by the Apple Macintosh in the mid 80s, the use of visual metaphors for human computer interaction became well established.

By the mid 80s, GUI-based personal computer applications were becoming available for desk-top publishing, CAD, and graphic design. A new group of users, non-computer-specialist design professionals, were now able to use these increasingly affordable and powerful tools. In order to accommodate this growing population of non-computer-specialist users, visual design applications borrowed heavily from their pre-digital predecessors, insulating the user as much as possible from the procedural internals of the system.

These GUI-based tools dominate the field of visual design software, and for good reason; such tools are both easy to use and powerful, providing non-computer-specialist visual designers significant improvements over non-digital design tools. Modern GUI-based visual design tools like Adobe Photoshop are mature and comprehensive, providing a reasonable tradeoff between interface
complexity and feature-set.

For all of their successes, conventional GUI-based design tools face a fundamental problem: no tool can be comprehensive, and with each additional feature the complexity of the user's interaction task increases. Inevitably there is a break-even point, where the benefit of adding additional features is outweighed by the disadvantages of increased user-interaction complexity. Improved user-interface design and careful organization may postpone this break-even point, but no GUI organization strategy can postpone it indefinitely.

In my opinion, that break-even point has already been reached, or even passed, in much commercially available design software. Ironically, the same GUI interaction techniques that made digital design tools so accessible at one level of sophistication make them frustrating and difficult to use at another. The complexity of tools which were once promoted for their ease-of-use now supports an entire industry of "how-to" publications. For example, a quick search of the Amazon.com online book store for books on Adobe Photoshop now produces 143 titles, mostly how-to books of one kind or other.

One way to address the problem of GUI interface complexity is by creating tools with procedural or scripting interface extensions. Procedural tools (tools with a procedural interface) are qualitatively different from conventional GUI tools, in that the user can customize and extend the capabilities of these tools to exactly match the problem at hand. The core functionality of a procedural tool can be simpler (and the GUI interface easier to manage) than a non-procedural tool, since flexibility of the tool comes from the procedural interface and not from a complex, feature-laden GUI.

The problem with procedural tools is that the use of a procedural interface is essentially a programming task. For many visual designers, programming is difficult and complicated; the primary reason for the development of the GUI interface was to avoid the use of programming or scripting. If there existed a way of providing the flexibility, power, and GUI simplicity of a procedural design tool without requiring the user to program, both problems (GUI interface complexity and scripting complexity) could be solved.

One way to view this is as an automatic programming prob-
lem. If it were possible to create the scripts for a procedural design tool simply, automatically, and graphically, without requiring the user to write the program themselves, then a nearly ideal compromise between interface complexity and tool flexibility could be achieved.

This research focuses on how such a tool might be created and used. More specifically, this thesis focuses on a new technique for the automatic generation of procedural design solutions, enhanced interactive genetic programming, and the ways in which the use of such tools changes the design process.

The framework for this discussion of the emergent design tools and process is a specific problem; the processing of digital images for visual design applications. I chose this problem for two reasons; first, the manipulation of images is a fundamental part of visual design, and second, conventional GUI-based tools for image manipulation have become very complex, and arguably are reaching the break-even point in feature-set vs. interaction complexity.

2 research overview

In the early 1980s I became interested in computer graphics and the process of making images using digital technology. At that time the graphics capabilities of personal computers were very simple and there was little in the way of sophisticated image creation or manipulation software available. I quickly exhausted the possibilities available on my Apple II+, and began to teach myself to program so that I could create new ones.

Later, as an undergraduate architecture student I studied photography, painting, and two and three dimensional composition using mostly non-digital techniques. By the time I reached graduate school I had decided that visual art and computer graphics were important areas of interest for me, and I spent three years studying visualization science at the Texas A&M Visualization Science program. In this context I studied the art and science of two and three dimensional image creation and manipulation using digital technology and traditional analog media.

One of the things that surprised me most about my education in digital media was just how literal the translation was from traditional "analog" tools and techniques to the new digital tools which,
by that time, had reached a considerable level of sophistication. The metaphors of photography and painting had been uprooted and transformed into the basis of interacting with complex systems of software that, at their core, were nothing like the process they emulated.

In many ways these traditional metaphors work very well as the basis for a graphical user interface, especially in the area of two-dimensional image creation and manipulation. However, conventional GUI-based tools limit functionality to those features provided by the graphical interface. No collection of pre-packaged features will ever be complete for all applications, which is why computer graphics systems programming was considered an important part of the visualization science curriculum; ideally, a professional working in the field of computer graphics should be able to create new tools from scratch if existing ones don’t provide a necessary capability.

These two extremes stand in stark contrast to each other; sophisticated GUI-based tools that insulate the user from the mechanics of the underlying process, or low-level programming environments for creating new design software from the ground up using (for instance) C++ and signal processing or graphics libraries. A few tools stand somewhere in the middle, providing a relatively high-level procedural interface to a complex process, such as Pixar’s RenderMan shading language which allows lighting, shading, and geometry-transforming effects to be specified in a C-like language, or Softimage’s Eddie image filtering and compositing system which uses a visual data-flow programming language to script the batch processing of images.

Procedural tools like RenderMan or Eddie provide a level of flexibility and power that are unmatched by their non-procedural counterparts, but are much more technically demanding to work with. They also tend to find application only at the high-end of the industry, since it is not expected that casual users will master the complexities of their use. At a lower level, general-purpose programming languages like C++ can be seen as the ultimate general-purpose procedural design tool, both in power and technical skill required for their use.

When I began working in the Aesthetics and Computation group at the Media Lab I was encouraged to focus on the creation of new
kinds of images and ways of image making. As a group, the ACG tends to focus on creating new kinds of visual experiences through procedural design using C++ and Java.

I became interested in creating a new language for procedural design with the goal of developing a syntax that was elegant and well-suited to exploring parallel, emergent systems. The language resulting from this experiment, Sol, is documented in [5], an unpublished technical report, and discussed at the end of this document in Section 12.

Sol proved to be a useful tool, but it did not make the process of procedural design any easier. Shortly after I began to work on Sol, I started thinking about how one could make procedural design more accessible to non-programmers. One approach is to abandon general-purpose programming languages and focus on simpler systems. This is the approach taken by Prof. Maeda in creating the language for Design by Numbers[19], a book intended to introduce procedural design to visual artists and designers without a programming background. Using a simplified or special-purpose language lowers the complexity of the programming task, but programming skill is still required.

I began to wonder if there was any way to harness the power of procedural design without requiring any programming skill on the part of the user; this is the problem of automatic programming. Automatically programming a computer may sound unlikely, but in fact there is a known means of accomplishing this task: genetic programming (GP)[17, 18, 2].

2.1 interactive genetic programming

Genetic programming is a search technique. The goal of applying search to solving a programming problem is creating code by searching for the solution, rather than by applying human intelligence to writing the code. This search takes place not in some library of known solutions, but in the abstract space of all possible syntactically correct expressions in a given language. Because the space is infinitely large and may be quite complex, this is generally a hard search problem; fortunately, genetic programming is a particularly powerful search technique. Through the application of genetic programming it is possible for a user to produce
programming solutions to problems without any understanding of programming whatsoever.

In conventional genetic programming, the user defines a fitness function which provides a measure of the quality of a trial solution. If the problem is well-posed, the specification of a useful fitness function is often easier than writing the code which solves the problem. When a problem is complex or ill-posed (most aesthetic problems fall in this category) there is often no obvious way to measure the quality of a solution other than by having the user evaluate the solution directly. Bringing the user “inside the loop” as the fitness function is the process of interactive genetic programming.

2.1.1 Karl Sims

Karl Sims, a Media Lab graduate, was the first to apply interactive genetic programming techniques to creating visual art on a computer[24, 25]. His work is very broad, covering the development of computational images\(^1\) to the evolution of amazing virtual creatures with complex forms and behaviors. The main drawback to this application of interactive genetic programming is that many generations of evolution may be required to achieve an interesting result, and (particularly in the case of images) this can be a very computationally intensive process.

Interactive genetic programming and its application to image filtering, is explained in more detail in Section 6.

2.2 enhanced interactive genetic programming

For small populations, conventional interactive genetic programming is a feasible approach, but for larger populations making side-by-side comparisons becomes difficult. My early experiments with genetic programming produced the GPI system for evolving procedural image filters through genetic programming. Work with the GPI system led me to wonder whether there was some way this process could be accelerated. Adding constraints or heuristics to the search process seemed like a reasonable approach, and

---

\(^1\) I use the term \textit{computational image} rather than \textit{procedural image} to avoid confusion with the term \textit{procedural image filter}. These terms are defined in Section 5.
this led me to the notion of enhancing the search process through automated preliminary fitness assessment.

The automated preliminary fitness assessment technique (APFA for short) allows the user to specify heuristics for the search in the form of a collection of preliminary fitness functions. Much like the operation of a conventional fitness function, each APFA function independently rates the population of solutions according to a pre-defined set of criteria. The combined rating of all APFA functions is then used to create a preliminary ranking of the solutions, and the user makes the final fitness assessment. The preliminary ranking produced by the APFA functions reduces the complexity of the user’s interaction task by allowing the user to focus attention on those solutions most likely to be good.

APFA works because it is possible to create fitness functions which evaluate part of an ill-posed genetic programming problem, even when it is not clear how to create a fitness function to evaluate the whole problem. This research suggests that even a small collection of simple APFA functions can make a large difference in managing the complexity of the interactive genetic programming process. APFA is described in more detail in Section 7.

2.3 emergent design

The use of enhanced interactive genetic programming to solve procedural design problems produces a design methodology called emergent design. As a case study, I applied the emergent design methodology to the problem of image filtering. The result is a software system called evolution++. The evolution++ system allows users to quickly evolve interesting procedural image filters through the use of APFA enhanced interactive genetic programming.

The emergent design process described in this thesis is an extension of the interactive genetic programming design process proposed in [24], although the application is different; Sims’ work focuses on creating computational images and artificial life, and this work focuses on developing image filters.

More importantly, the enhanced genetic programming process described in this thesis provides significant improvements in flexibility and power over the non-enhanced process described by Sims.
Through the use of APFA, the complexity of the user's interaction task is decreased, making larger searches possible. Larger searches improve the speed and likelihood of convergence, allowing for results in fewer generations and with less computational expense.
3 emergent design — a hypothetical example

This section provides a hypothetical example of a working emergent design process; this is important to provide the reader with a feel for what this research is attempting to achieve, and what the emergent design process could ultimately provide for the artist or designer who use this process.

In describing the Cassero Bella project, I use terminology I haven’t yet defined; much of it, I hope, is understandable from the context. In particular, if the reader is not familiar with the basics of genetic programming, I suggest skipping ahead to Section 6. Likewise, the APFA (Automated Preliminary Fitness Assessment) framework is explained in Section 7. The best approach to reading this section may be to skip ahead and back as necessary.

The tools described in this section go beyond the software implemented for this thesis, but this is a difference in refinement and integration, not a qualitative difference in the technology. In particular, the interactive video filtering tool described here is in all important respects identical to the evolution++ system, with some interface enhancements.

The texture-matching APFA system is more advanced than anything yet implemented, but is included to show what a sophisticated realization of the emergent design process could be capable of.

3.1 the Cassero Bella Center

The setting is a digital design studio, and the time is the near future. A group of artists and designers are collaborating on a multimedia project, one which will combine still and moving images, sound, two-dimensional and three dimensional interactive computer graphics.

The project is an artistic interpretation of a historical reconstruction project, commissioned by the town of Cassero Bella\(^2\) in central Italy. The town is a classic Tuscan walled city, constructed around an ancient medieval hill-top castle. This castle, which is now in ruins, is to be partially reconstructed and turned into the

\(^2\)Cassero Bella is a fictitious town, based on Castiglion Fiorentino and a number of other Tuscan hill towns I visited in Italy.
Cassero Bella Center, an art gallery and cultural center for Cassero Bella and the surrounding region. The reconstruction is a long-term project that will be accomplished in stages over a period of years.

The goal of the multimedia project is to produce an interactive kiosk to be located on site in the cultural center. The kiosk must be done quickly, since it is the centerpiece of the ground-breaking ceremony to be held in a few months. The project is difficult because time is short and the commission is ambitious: The kiosk should provide a presentation of the town’s history, explain the reconstruction process and allow visitors to virtually explore the site in different phases of reconstruction while portions of the real site are inaccessible.

3.2 the technical challenges

The project has many parts, including 2D and 3D modeling, animation, video production, sound, and interaction design. The result must be visually compelling and well-integrated. One of the particular challenges is that the final presentation will be constructed interactively and on-the-fly, combining many different visual elements.

3.3 the emergent design process

A design team is dispatched to Cassero Bella to photograph and record the site in its current state, and video-tape interviews with local residents and historical authorities. At the same time, another group works on digitizing the drawings, artwork, and architectural reconstruction plans provided by the Cassero Bella Center committee.

While the remote team is still on-site, digital photographs of the castle and its surroundings are sent back to the studio. The designers there are already constructing the three-dimensional model which will be the virtual Cassero Bella Center. The model geometry is largely done (much of it was digitized from the architectural plans and drawings), and the next step is applying textures to give the castle model the look of the native stone. These textures must be very good, since the virtual model must integrate seamlessly with real photographs in the virtual reconstruction.
In the past, the modelers would have developed a simple non-procedural texture by hand, or used a generic stone texture from a texture library; developing a sophisticated procedural texture to exactly match the idiosyncrasies of the castle masonry would be too time-consuming for this application. Now, using new emergent design tools, the designers quickly evolve a procedural texture to exactly match the exterior of the castle.

3.3.1 a sophisticated emergent design system

For the designer, evolving the castle texture is made particularly simple through the use of a sophisticated texture-matching emergent design system. This enhanced genetic programming system uses a sophisticated collection of APFA functions specifically designed to help evolve procedural textures to match the qualities of real objects, using digital photographs as a reference.

In order to use this system, the designer must first match the virtual lighting conditions of the computer model to the real outdoor photographic lighting conditions. This is not a matter of guess-work, since the geographic location, time of day, and weather conditions under which the photographs were taken are known. Now, the designer asks the texture-matching emergent design system to match a lit, rendered view of the castle model to photographs of the castle, each taken from different angles.

The designer tells the system what type of digital camera was used and an initial guess about the location and orientation of the photographer for each of the images. The emergent design system uses the three-dimensional model of the castle and a knowledge of the camera’s optics to refine this guess, quickly determining the camera’s exact location and orientation for each photograph. A virtual camera with same optical properties is positioned to match the location of the real camera in each photograph.

Then the interactive search for the castle texture begins. The emergent design system randomly generates thousands of procedural textures; each texture in sequence is applied and the textured castle geometry is quickly rendered at low resolution using each of the virtual cameras. The system examines the result of the renderings and assigns the filter a ranking based on the quality of the match for each virtual camera view. The resolution of each ren-
dered image is very low so the whole process takes only a few seconds.

The modeler asks to see the top fifty results, as ranked by the APFA functions. Initially, each texture is represented iconically by its low-resolution renderings. By clicking on a texture icon, the designer selects a texture to apply to the castle model and rendered interactively at full resolution. Because the initial population was large and the matching criteria were precise, most of the top-ranked guesses produce rendered castles which look something like the photographs, at least from a distance.

The designer decides the APFA functions are doing a good job with the ranking, and selects the top-ranked one hundred filters to be parents of the next generation, this time telling the system to match at a higher resolution. A thousand children are produced by recombining the one hundred selected filters, and again the APFA functions rank the results, which the system again displays for the designer.

After five generations (a process taking thirty minutes, of which less than two minutes was spent waiting for texture generation and ranking), the results are starting to look very good. The top-ranked procedural textures are matching the masonry pattern and complicated rough texture of the stone very well, but the color isn’t quite right. The designer adjusts the weights of the APFA functions to emphasize color matching in succeeding generations.

Three generations later seven textures are produced that almost perfectly match the photographs. The designer again adjusts the APFA function weights to make computational complexity the most important criterion and re-ranks the results. Surprisingly, one of the filters is much simpler than the others. The designer chooses this filter, and the task of texturing the castle is done.

### 3.3.2 real-time video filtering

As the other pieces of the presentation are being assembled, the lead designer is faced with the challenge of tying all the pieces together in a way which will match the narrative of the historical story. Her plan is to have several top-level narrative threads selectable by the user, and at the same time allow the user to freely explore the virtual castle. Some spaces in the castle will have dif-
ferent multimedia content for each narrative thread, but there isn’t
time to provide special content everywhere. The whole feel of the
presentation must change based on the mood of the discussion,
from the joy and optimism of the Renaissance to the despair of
war and plague.

The sound-track will help, but the lead designer feels that more
is needed. She would like to expand the use of theatrical lighting,
but for reasons of historical accuracy, day-time scenes will be lit
using an approximation of natural light. The kiosk hardware has
excellent graphics capabilities, so she decides to use a sequence
of real-time video filters to interactively “post-process” the entire
multimedia experience.

Not all of the multimedia content is ready, but by this time
a good sampling is available, and the story-board of each of the
narrative threads is done. Using emergent design tools, the lead
designer will evolve a scripted sequence of real-time video filters
for each of the narrative threads.

The texture-matching tool was based on a tightly integrated
and specialized collection of APFA functions. The emergent de-
sign video processing application is much more general, and this
time there is no pre-packaged collection of APFA functions ex-
actly suited to the task. Rather, the lead designer chooses a from a
wide selection of simple APFA functions, each of which measures
a potentially useful video filter property.

Because the video filters must run at interactive rates in paral-
lel with the rest of the multimedia presentation, one of the APFA
functions is a complexity measure, giving simpler, more efficient
filters better ratings than more complex ones. The filters should be
at least somewhat naturalistic, so a function is chosen that penal-
izes geometric distortion, scaling, and warping. Likewise, another
function is chosen that favors filters which work with all three
color components (hue, saturation, and value) of the source im-
age.

Using sample content from the first sequence of the first story-
board, the lead designer begins the filter evolution process. She
is not starting entirely from scratch; there is a collection of video
filters she has used before. None of these filters is likely to be
ideal, but some may be close. This filter collection forms the basis
of the initial population, to which hundreds of randomly generated
filters are added.

Each APFA function rates the initial population separately, resulting in multiple fitness assessments for each filter. By default, the video processing system ranks the results by overall best score, treating the assessment of each APFA function equally. The lead designer asks to see the fifty top-ranked filters, and the evolution interface complies by displaying each of the filters as a small, animated icon for side-by-side comparison. Each icon is a scaled-down version of the corresponding filter applied to the full-size input video stream.

All fifty of the displayed filters are running in parallel on the sample multimedia content, the unfiltered version of which is displayed at the bottom of the screen for comparison. The designer selects particular filters with the mouse to see full-sized renderings. Some of the top-fifty filters chosen by the current APFA configuration are interesting, but many are not.

The lead designer decides that the current APFA weighting is not producing a good ranking. She pulls up a box of sliders and adjusts the weights, the filter rankings shifting as the weights are adjusted. The results are still unsatisfactory, so she adds an additional APFA function (written by the designer herself) that penalizes color distortion.

Now satisfied with the agent population and weighting, the designer selects several dozen filters to be parents of the next generation, saving one of them separately for possible later use. A new generation is produced, and the process is repeated. After five generations, the designer has produce three strong candidate filters for the first story board sequence. This process is repeated for each sequence.

Having selected trial filters, the lead designer plays with several alternative sequences for each story board, and scripts the transitions between them. Now the sequences can be integrated with the rest of the multimedia content for the narrative threads, which has been assembled using a combination of conventional and emergent design tools. In many cases, the emergent design tools produced multiple interesting design solutions. In many cases these multiple alternatives are incorporated into the final multimedia experience, adding additional depth and variety.
4 creating images

This century has seen an explosion in the number of media available for image making. Painting, photography, film, video, screen-printing, lithography, xerography, and collage are only some of the possibilities available before the advent of computer graphics in the 1960s. Computer graphics has grown up in an image-technology-rich environment, and it is natural that image processing and manipulation would be an important graphics application. It is also natural that digital image processing tools borrow much from their analog predecessors, both in form and user-interaction methods.

4.1 digital image manipulation

This thesis focuses on systems and techniques for digital image processing in a design context. Digital image processing is the manipulation of discrete, sampled representations of two-dimensional images using a digital computer[14].

It is important to note that sampled image data is only one way of describing a picture in a digital context. For instance, the PostScript page description language[1] uses arbitrary geometric primitives to compose pictures which can then be displayed at any resolution. However, sampled images are very important since image sampling is the most general way of getting images into and out of the computer. Even the geometric primitives of a PostScript document must be scan-converted into a sampled image raster for output on a laser printer or photographic type-setting machine.

4.1.1 history

Early computer graphics research focused on the technical problems of representing and displaying graphical information, and of interacting with the user in a graphical manner[6]. From the very beginning, graphics researchers looked to other, non-digital graphical tasks as the basis for interaction.

One of the earliest important developments in the field of computer graphics software was the Sketchpad system developed by Ian Sutherland in the early 1960s[28]. As the name implies, this system allowed the user to work with graphical objects using a
drawing metaphor. By allowing the user to work with lines and shapes using a light pen, the Sketchpad system laid the groundwork for WYSIWIG\(^3\), direct manipulation, and other graphical user interface techniques.

The high cost of graphics hardware confined early computer graphics to high-end engineering and military applications. Most early systems employed vector displays that rendered geometric shapes as a series of lines or curves swept out by deflecting a cathode ray across a phosphor screen, much like an oscilloscope trace. Vector displays were used because the memory required to store an entire sampled image for output (a raster) was extremely expensive. Digital image manipulation, which relies on raster-graphics hardware for display and interaction, was not economically feasible for most design applications until the advent of cheap raster graphics displays and personal computers in the early to mid 1980s.

4.1.2 present technology

Image processing software for design applications generally falls into one of two categories. Traditional image editing systems afford a GUI-based image-by-image editing process through a painting or photography metaphor. These systems generally require the user to interact with the image through a mouse or graphics tablet, using virtual tools inspired by some physical process like sketching, masking, blending, etc.. Procedural image editing systems (like the previously mentioned Eddie, a high-end compositing and filtering system produced by Softimage) provide a scriptable interface for filtering images and image sequences. This interface may be graphical (Eddie uses a visual data-flow programming language) but the process of image manipulation is specified as a sequence of general image-manipulation operations using a scripting language.

The first type of tool provides a high degree of control over individual images, but generally provides a limited means of generalizing operations across a sequence of images. The second type

\(^3\)What You See Is What You Get: an HCI technique describing systems in which the user works with an interactive visual representation that is essentially identical to the output of the application.
of tool provides less specific control over individual images but greater facility to generalize operations across multiple images.

Image editing systems which fall into the first category (Adobe Photoshop, for example) recast digital image processing as a traditional design activity. By doing so, these tools provide traditionally-trained design professionals and others comfortable with the interface metaphor a conceptual framework for interacting with the system. In addition, the interface metaphor usually extends to the application domain as well; for example, Adobe Photoshop provides its users with a conceptual framework based on photographic darkroom metaphors, and generally serves the analogous role of image-retouching in the design process. The disadvantage of a traditional GUI-based system is the tradeoff between feature-set and interaction complexity discussed in Section 1, and the lack of generalization discussed above.

Batch processing and scripting tools typically require much more technical expertise, and represent the process of image manipulation in abstract, procedural terms. The advantage of such an approach is generality; rather than describing actions in relation to a particular image, these tools allow the user to describe a general process which can be applied to any image. It is then up to the tool to execute the operations on the images. What the user gives up in control over the specific details of any one image they gain in the ability to work with large groups of images in a systematic, automated way.

This second type of tool is what I will call a procedural design tool, in that it allows designers to specify a procedure for accomplishing a design task rather than providing a gestural, WYSIWYG graphical interface. Using such a tool is essentially a programming task, and the greater the flexibility of the tool the more technical expertise (in a programming sense) is needed to use it effectively. From this point of view, the ultimate procedural design tool is a general purpose programming language, since it provides maximal flexibility and power, but also requires the highest level of technical expertise. The use of procedural design tools is a uniquely digital possibility, and the idea of a procedural design solution, a design solution that takes the form of a procedure or program, provides the foundation of the emergent design process.
5 image filtering

This section describes a framework for image processing using procedural image filters. This framework has the advantages of being general, powerful, and well-suited to both procedural design and genetic programming applications. In this discussion, it is assumed that the format for both input and output images is spatially-sampled color or luminance information; a raster, in other words.

The pixels which make up a digital image represent discrete samples of a continuous two-dimensional image. At the most general level, an image filter is an algorithm which specifies the transformation of a source image to produce a destination image. This typically involves reconstruction of a continuous two-dimensional signal from the sampled source data, the re-sampling of this two-dimensional signal by the filter, and the construction of new sampled pixel values for the destination image.

The discussion of image filtering that follows begins with the notion of a simple, discrete pixel-to-pixel mapping for gray-scale images. From there, I introduce the idea of procedural image filters, signal reconstruction, resampling, and color.

5.1 discrete geometric transforms

In order to create an image filter, it is necessary to formally specify the mapping or transformation of a source image that will produce the desired destination image. The simplest possible framework for such an operation is a discrete, pixel-to-pixel forward mapping. There are many different ways to specify such a mapping; in the case of simple geometric transforms, mappings are often specified as a transformation matrix which maps the coordinates of the destination image into the source image.

If a geometric transformation is implemented as a discrete forward map, meaning that each pixel in the source images is mapped into a pixel in the destination image, then transforms which warp or scale the image are likely to cause gaps and other unwanted artifacts in the destination image.

A better solution is to specify a geometric transform as a discrete pixel inverse map\(^4\), where each pixel in the destination image

\(^4\)Inverse mapping does not solve all image-artifact problems; see the discus-
is mapped into zero or more pixels of the source image. Constructing this type of inverse-mapping geometric transform can be done by inverting the forward-mapping transformation matrix.

5.2 a simple procedural image processing framework

A general image filtering framework must allow more than geometric transforms; it should also be possible to create image filters that alter the color or luminance information of the source image. This can't be done through simple transformation matrices, but it can be done through a procedural image filtering framework.

Imagine a discrete pixel inverse-mapping gray-scale image filter in which the value (luminance) of each pixel in the destination image is specified as a function of the value of zero or more pixels in the source image. Assuming that the source image and destination images are of the same size, then each pixel in the destination image corresponds to exactly one pixel in the source image. By assigning each pixel an \( x, y \) coordinate and using the notation \( V_s(x, y) \) to refer to the value of the pixel \( x, y \) in the source image, the identity filter (the filter which produces a destination image identical to the source image) could be specified as

\[
V_d(x, y) = V_s(x, y),
\]

where \( V_d(x, y) \) is the value of pixel \( x, y \) in the destination image.

So far we have only demonstrated an abstract way of copying the contents of an image, which might better be accomplished by other means. Things become interesting when it is possible to perform operations on pixel values more complex than assignment. For instance, if multiplication is allowed then an excessively bright image could be made uniformly darker with the filter

\[
V_d(x, y) = \alpha V_s(x, y),
\]

where \( 0 \leq \alpha < 1 \). This raises the question of pixel value representation. If discrete quantities are used to represent value \( (0 \leq s < 1) \), then reconstruction and aliasing below.

\(^3\)This is a geometric correspondence, not a functional relationship. Any pixel or pixels of the source image may be used to calculate a pixel in the destination image. A one-to-one correspondence is a convenience for describing operations in the discrete pixel framework; the continuously resampled image filtering framework described later allows source and destination images to be of any size.
i ≤ 255, i ∈ Z, for instance), than scaling by α will require type conversion and rounding. Converting each source pixel value to a floating point number in the range 0 ≤ v < 1 before processing allows filters to be written with fewer assumptions about the input data format.

For values of α > 1, the same filter expression could be used to make a dark image brighter. But what happens if αV_s(x, y) ≥ 1? We could clamp the value at 1 - ε where ε is some small value, but that may not be the best solution for all applications. A better approach is to give pixel value periodic boundaries, so that a pixel value outside the legal range “wraps around” to a value 0 ≤ v < 1, and to provide a clamp function in our filter toolkit to prevent this wrapping if desired, e.g.

\[ V_d(x, y) = \text{clamp}(αV_s(x, y)) \]

where

\[ \text{clamp}(u) = \begin{cases} 
0 & \text{if } u < 0 \\
1 - ε & \text{if } 1 ≤ u \\
u & \text{otherwise}
\end{cases} \]

So far we have demonstrated filters in which a single pixel in the destination image is the result of operations on a single pixel in the source image. This is not a limitation of the filter architecture. For instance, a simple low pass filter could be specified as

\[ V_d(x, y) = \left( V_s(x - 1, y - 1) + V_s(x, y - 1) + V_s(x + 1, y - 1) + V_s(x - 1, y) + V_s(x, y) + V_s(x + 1, y) + V_s(x - 1, y + 1) + V_s(x, y + 1) + V_s(x + 1, y + 1) \right) / 9, \]

which averages the value of pixel \( x, y \) and its eight nearest neighbors. This filter works fine in the center of the image, but what happens when it is applied at the edge? For instance, \( x + 1 \) might specify a location off the right edge of the source image. One way to solve this problem is to decide that any samples taken outside the image return a zero value, but this is arbitrary and makes no reference to the sampled image data. A better way to handle the situation is to assign the image periodic boundaries, or a toroidal mapping; this way, any \( x, y \) location refers to sampled pixel data.

By using a floating point representation for pixel value, and assigning periodic boundaries to our images and to the calculated
values in those images, we have gone a long way towards producing a usable filter framework. Three primary issues remain unresolved: the problems of discrete pixel data and resampling, color, and the representation of our filters.

5.3 discrete data and signal reconstruction

A digital image is a discrete representation of what our eye regards as a continuous two-dimensional signal. From the point of view of the computer, the data representing the pixel values of the image is simply a binary string. The organization of this data into a two-dimensional raster and the rendering of this raster as a color image provides the viewer with a visual reconstruction of the original signal. It is important to note that this visual reconstruction ultimately happens in the human eye and mind, not in the computer.

Image filtering requires more than a visual reconstruction of the image; after all, our filters exist within the computer, not the human eye. In order to do image filtering well, we must provide our filters with a mathematical reconstruction of the continuous signal. This continuous reconstruction is necessary, because the original sampled pixel values may not align with the samples needed by any particular filter.

This problem is most clearly illustrated by a simple image scaling operation. Imagine a filter which scales an image by a factor of two in both width and height:

$$V_d(x, y) = V(x/2, y/2).$$

Clearly, the filter needs to sample the image between the original samples, at fractional $x$ and $y$ values. Since we have no sampled data for these fractional pixel locations, this requires reconstructing the original signal in order to resample it.

5.3.1 continuous coordinates

There are many possible ways to reconstruct the signal for resampling, some of which are more accurate than others. Before we can use any of these techniques, we must allow for fractional $x$ and $y$ values. However, once we reconstruct the signal the original meaning of $x$ and $y$ as the location of discrete data samples
loses its importance; we can now resample the signal anywhere, in a continuous \( u, v \) image plane.

We could use \( u, v \) coordinates that correspond to the \( x \) and \( y \) sample locations of the original image, but this is inconvenient if we want to create filters which will operate the same way on source images of different resolutions. For instance, examine the following sinusoidal image warp filter:

\[
V_d(u, v) = V_s(u + \sin(\alpha u)/10, v).
\]

If this filter is applied to two images, where the first image is twice the resolution of the other, the alpha value of the filter for image one, \( \alpha_1 \), must equal one half the alpha value of image two in order to achieve the same effect, \( \alpha_1 = \alpha_2 / 2 \).

In order to make all image filters operate the same regardless of input image resolution, it is necessary to normalize \( u, v \) coordinate values (make the range the same for all image resolutions). Any scale could be used for this normalization, but in practice it proves convenient to normalize these coordinates such that the values \( 0 < u < 1 \) and \( 0 < v < 1 \) map to the full range of \( x \) and \( y \) values in the source and destination images. As before, we assign periodic boundaries to \( u \) and \( v \) so values outside the zero-one normalized range “wrap” back into the resampled image plane.

### 5.3.2 resampling

With the use of continuous image-plane coordinates comes the ability to specify samples between the original sampled image data. This capability is not useful unless we have some means of reconstructing the signal between the sampled data points. The sampling theorem states that in order to reconstruct a signal without error, it is necessary to sample it at a rate greater than twice the highest frequency of interest, otherwise known as the Nyquist frequency. The surprising implication of this statement is that it is possible to perfectly reconstruct an adequately sampled signal; one way to do this is to perform a discrete Fourier transform on the sampled data and then expand the result in terms of sine and cosine functions.[14, 7]

Such a transformation and expansion is relatively computationally expensive. For most image filtering applications, we are
far less interested in perfect reconstruction than a fast reconstruction which is good enough; clearly, this criterion is application dependent.

The fastest reconstruction is to use the value of the nearest sample (zeroth-order pixel replication), which is equivalent to reconstructing the signal using step functions. For applications like real-time video processing where the signal tends to be noisy and performance is at a premium, this may be sufficient. A slightly more expensive technique is linear interpolation, which is equivalent to drawing straight lines between sampled data points. Linear interpolation provides a much smoother reconstructed signal than the nearest-sample technique, and is sufficient for many purposes [14]. Higher-order reconstructions are possible, but the increased smoothness comes at the price of significantly increased computational expense.

5.3.3 aliasing

A poor reconstruction introduces errors, which are passed on to the filters as they resample the signal. Another source of error is the resampling process itself; a perfect reconstruction is not useful if a filter fails to resample it at a sufficiently high frequency. Sampling and reconstruction errors are collectively known as aliasing, which can introduce unwanted visual artifacts into the resulting filtered images. For an extended discussion of image sampling, reconstruction, and aliasing see [14].

5.4 color

Color is a very important visual design issue; the appropriate use of color is one of the factors which delineates good and bad design. Providing the designer with a means to effectively use color is an important part of any visual design tool.

5.4.1 the RGB color space

A common system for representing color in digital images is the RGB color space. The RGB color space represents color as a linear combination of red, green, and blue light. The RGB space is conceptually simple and easy to work with at the software level,
The RGB color space. since most raster graphics systems use an RGB encoding at the hardware level.

The RGB color space provides a surprisingly poor system for working with color in a design context. Human color sensation is based on the relative levels of stimulation of three types of color sensors (cones) in the eye. Although these color sensors have sensitivity functions roughly corresponding to the red, green, and blue portions of the spectrum, the perception of color is a more complex phenomenon (see [6] for an in-depth discussion of color sensation and color spaces).

5.4.2 the HSV color space

The perceptual properties of hue, saturation and value have been used for years by artists and designers to describe color, and are more easily understood than a simple additive combination of red, green, and blue. The HSV color model was developed to provide a user-oriented color model for computer graphics that is more compatible with traditional artistic color theory.

The HSV color space parameterizes color in terms of hue, saturation, and value, where hue specifies a distance around the perimeter of a color wheel, saturation specifies the distance from the gray center, and value is brightness. Picking colors in HSV space is generally easier than in RGB space. More importantly, general operations on color in HSV space are easier to understand and control than in RGB space[6].

5.4.3 color image filtering

It is simple to extend the framework we have developed for filtering gray-scale images to filtering color images, particularly if the HSV color space is used. An HSV image filter can be specified as three separate filter expressions, one each for hue, saturation and value. (This is also true for RGB image filters.) For example, the HSV identity filter can be specified as

\[ H_d(u, v) = H_s(u, v) \]
\[ S_d(u, v) = S_s(u, v) \]
\[ V_d(u, v) = V_s(u, v) \]
where $H$ and $S$ are the hue and saturation sampling functions, and $V$ retains its previous meaning as the value sampling function. A vector notation could also be used, but mixing vectors and scalars proves inconvenient for genetic programming applications; see Section 6.2.3 for more details.

5.4.4 the choice of HSV over RGB

One reason to choose the HSV space over RGB for image filtering is that an HSV filter may be constructed from a gray-scale filter simply by adding two expressions, as we have shown. Likewise, any HSV filter may be used as a gray-scale filter simply by ignoring the hue and saturation terms.

A better reason to choose HSV is that filtering color in HSV space is easier to understand, and effects are easier to achieve. This is important for procedural design applications when humans are writing filter code, but it is also important for genetic programming applications.

**HSV as a perceptual diagonalization of color** Transforming to an appropriate basis is extremely important for any type of modeling activity, and can drastically simplify the modeling process[7]. Meaningful filters are *simpler* in HSV space than RGB space. The space of simpler expressions is smaller than the space of more complex expressions, and this makes simpler expressions easier to find using genetic search. Another way of saying this is that we are interested in filters which create *perceptual* changes in images. Such filters are most easily represented in a perceptually meaningful color space that diagonalizes (decouples) the perceptual properties of color.

5.4.5 the complexity of HSV vs. RGB filtering

Common tasks in image manipulation for design applications include making an image darker or lighter, making the colors more or less intense, and shifting the color of an image towards a particular hue (making an image more blue-green, for instance). All of these tasks are simpler to describe, and easier to accomplish in an HSV color space. The following paragraphs provide side-by-
side comparisons between HSV and RGB filters that accomplish the same tasks.

**Making an image darker or lighter** In this example, the filter adjusts the brightness of an image by a scale factor $\alpha$. If $\alpha$ is greater than one, the image becomes brighter. If the value of $\alpha$ is less than one, the image becomes darker. In HSV space, this can be specified by a single expression,

$$V_d(u,v) = \alpha V_s(u,v),$$

assuming that the unspecified hue and saturation expressions are the identity filter. The corresponding RGB filter consists of three expressions,

$$R_d(u,v) = \alpha R_s(u,v)$$
$$G_d(u,v) = \alpha G_s(u,v)$$
$$B_d(u,v) = \alpha B_s(u,v),$$

where $R$, $G$, and $B$ represent the red, green, and blue sampling functions, respectively. Note that all three components of the RGB image must be altered to change its brightness, but only one component of the HSV image must be changed. This is because brightness (value) is a basis vector of the HSV space, but is not orthogonal to any of the RGB basis vectors.

**Changing color intensity** Making colors more or less saturated is another common filtering task. Again, in HSV space changing color saturation by a scale factor $\alpha$ involves a single expression,

$$S_d(u,v) = \alpha S_s(u,v),$$

where values of $\alpha$ less than one make colors less saturated and greater than one make them more saturated. In RGB space this task is significantly more complex. First, an average intensity $I$ must be calculated,

$$I(u,v) = \frac{R_s(u,v) + G_s(u,v) + B_s(u,v)}{3}.$$
Second, the difference $\delta$ from the average for each component must be calculated, and scaled by $\alpha$.

$$\begin{align*}
\delta_R(u, v) &= \alpha (R_s(u, v) - I(u, v)) \\
\delta_G(u, v) &= \alpha (G_s(u, v) - I(u, v)) \\
\delta_B(u, v) &= \alpha (B_s(u, v) - I(u, v)).
\end{align*}$$

Third, the scaled $\delta$ values for each component are added back to the average to produce the new color values for each component,

$$\begin{align*}
R_d(u, v) &= I(u, v) + \delta_R(u, v) \\
G_d(u, v) &= I(u, v) + \delta_G(u, v) \\
B_d(u, v) &= I(u, v) + \delta_B(u, v).
\end{align*}$$

Again, the difference in complexity is the result of a difference in orthogonality; saturation is a basis vector of the HSV space, but not orthogonal to any of the RGB basis vectors.

**shifting color** The type of color shifting effects that are easy to achieve in HSV space are very different from those in RGB space, and vice versa. This is because linear interpolation in HSV space is not generally equivalent to linear interpolation in RGB space, as is noted in [6].

A simple additive combination of color, like mixing paint, is easiest in RGB, and can be described by

$$\begin{align*}
R_d(u, v) &= (1 - \alpha)R_s(u, v) + \alpha R_t(u, v) \\
G_d(u, v) &= (1 - \alpha)G_s(u, v) + \alpha G_t(u, v) \\
B_d(u, v) &= (1 - \alpha)B_s(u, v) + \alpha B_t(u, v),
\end{align*}$$

where the target color is specified by the $R_t$, $G_t$, and $B_t$ functions and $\alpha$ is an interpolation weight. This type of color mixing will change saturation and value as well as color.

In HSV space, it is possible to shift hue while leaving saturation and value unchanged. An HSV hue-shift filter may be specified as

$$H_d(u, v) = (1 - \alpha)H_s(u, v) + \alpha H_t(u, v)$$

where $H_t$ is the target hue and $\alpha$ is a weighting parameter. The RGB-space equivalent of the HSV hue-shift is quite complex, and
best described in terms of a conversion to HSV space for the interpolation, and then a conversion back. Because this conversion involves a number of special cases, it is best described as an algorithm, which the curious or highly motivated reader may find in [6].

5.5 filter representation

Up until now we have specified the image filter expressions in a general mathematical notation. This is fine for explanation, but for use in our image filtering system the filters must be represented procedurally. There are many possible choices for the syntax of this representation, and were all image filters written by hand the choice would be mostly a matter of taste. Since the goal of the procedural filtering framework is to enable image filters to be found through genetic programming, a good choice is a Lisp syntax[15, 17, 18], since Lisp expressions are syntactically simple and easy to manipulate. The justification for choosing Lisp will be explained in detail in Section 6.2.3. Once again, here is the identity filter, this time represented in Lisp notation:

```lisp
;; The HSV identity filter
;; as a list of three LISP expressions.
((hue u v) ; the hue expression
  (saturation u v) ; the saturation expression
  (value u v)) ; the value expression
```

More complicated filters are also easy to represent in LISP notation once the prefix notation becomes familiar. Here is the LISP version of the sinusoidal warp filter from Section 5.3.1 (this is the actual LISP expression used to make the example image):

```lisp
;; Sinusoidal warp filter,
;; gray-scale (one expression)
;; version.
((value (+ u (* .1 (sin* (* 12.28 v))))) v))
```

Here is one last example; a relatively simple, yet non-intuitive image filter which produced the filtered results shown here:
This filter points out the main disadvantage of using procedural design tools; these tools can be very powerful, but not easy to understand. As it turns out, it is not necessary to understand how procedural image filters work in order to create and use them; the next section addresses this issue.
6 genetic programming for image processing

The procedural image processing framework discussed in the previous chapter requires significant programming expertise and an understanding of image processing fundamentals in order to develop new filters, which is the essential problem of procedural design. Ideally, there would be some way to extend this framework to allow almost anyone to use it without requiring image processing knowledge or programming skills. This appears to be a hard problem, almost a Zen koan: how is it possible to solve programming problems without programming? This is the problem of automatic programming.[17]

6.1 automatic programming

One solution is for the author of the system to provide the user with a wide variety of filters in the hope that one of the pre-written filters will work well enough for any particular filtering application. Such a system could be useful, but with a small number of filters would not be very flexible.

In order to provide a non-programming user the flexibility a filter-writing user possesses, the pool of pre-written filters would have to be very large. At that point, the problem becomes finding an applicable filter within a large filter population, most of which are not likely to be well-suited to the task at hand.

At first glance this may seem like an unlikely avenue towards a practical solution, but recasting filter programming as a search problem has an interesting result: if the population of filter candidates becomes very large, than the user is no longer looking for the single best filter, but rather for any filter which is good enough. And if the population is large enough, there may be any number of good filters to be found.

Strangely, the truly hard non-linear search problems are not the largest; in fact, there is a critical problem size between large and small in which the hardest problems lie[7]. This transition in complexity is fairly sudden and has many of the properties of a phase-transition[8].

---

6 A phase-transition is a sudden shift in the properties of a physical system resulting from a small change in one parameter, like liquid water freezing into
The transition in search difficulty from easy, to hard, and back to easy as a function of problem size is not an obvious result, nor is it easy to explain without resorting to complex mathematical arguments. Informally, as the number of degrees of freedom grows, the number of potential solutions grows exponentially. In a low-dimensional space, there are only a few combinations to check, and these can be exhaustively searched. In a high dimensional space, the number of potential solutions is huge, but there are also many ways of solving the problem. In the middle ground, there are too many possibilities to check exhaustively, but few ways of solving the problem. An extended discussion of this phenomenon may be found in chapter 13 of [7].

The result is that beyond a certain threshold, finding a solution in a higher dimensional space may actually be easier than finding a solution in a lower dimensional one. By extension, conducting a search in the infinite dimensional space of all possible filter expressions might actually be much easier than trying to find a particular member of a large, finite population, if the proper technique is employed.

6.2 genetic search

Genetic search is a term used to describe a family of highly parallel search techniques which are particularly well suited to finding solutions to complex problems. In genetic search, the search process mirrors very closely the process of evolution and natural selection in nature; an initial population of trial solutions is allowed to compete for reproduction under the selective pressure of solving the problem. In each generation, the “fit” solutions (those that solve the problem better) out-compete the worse solutions, and have a better chance of passing their genes on to the next generation. This results in successive generations of better and better solutions.

Genetic search can be broken into two distinct sub-categories: genetic algorithms (GA) and genetic programming (GP). Since genetic programming is essentially a generalization of genetic algorithm search, genetic algorithm search will be presented first. The discussion of GA and GP that follows is somewhat informal; a detailed explanation of the formal theory (especially as it pertains to

ice as temperature drops from 0.5° to -0.5° Celsius.
GP) can be found in [17, 18, 2].

6.2.1 genetic algorithms

In genetic algorithm search (GA), potential solutions are coded as sequences of discrete symbols, or genes. The goal of the search process is to find a combination or sequence of genes that will solve (or approximately solve) the problem. The \textit{parameterization} of this coding (what the genes mean) is very important, since it defines the space in which the search takes place. The representation for the genes is essentially arbitrary; any convenient representation allowing crossover, such as ASCII strings, will do. This is not true of genetic programming, which is discussed later. The general framework for a genetic algorithm search is as follows:

1. An initial population of strings is randomly generated.
2. Each string in the population is evaluated with a problem-specific fitness function, resulting in a fitness score.
3. The solutions with the best fitness scores are recombined to produce a new population.
4. Steps two and three are repeated until a sufficiently good solution is found.

\textbf{generating the initial population} In order to find a good solution, it is necessary to start with an initial population which is large and diverse enough to provide a reasonable sampling of the search space. “Large enough” is a relative term which depends on the number of degrees of freedom and how hard the problem is. Diversity is achieved by drawing the initial population from a random distribution, ideally one which is evenly distributed over the search space.

\textbf{applying the fitness function} The idea of a fitness function is based on the observation that it is almost always easier to evaluate the quality of a trial solution than to analytically or procedurally solve a non-linear problem (which may be impossible). The criteria that a particular fitness function uses depends entirely on the problem; often, search problems are extremization problems,
e.g., energy minimization. In this case, genes coding for a lower energy state result in a better fitness score. In order to be useful in GA applications, a fitness function must be able to distinguish among better and worse solutions, not simply recognize the one or two which solve the problem perfectly.

Because the fitness score is a measure of how well a particular trial solution solves the problem, finding a trial solution with a good enough fitness score means the problem is solved and the search can stop. “Good enough” does not mean best; if a problem is hard enough to require a genetic search technique, finding the single best solution, or knowing that you have found it, is often impossible. Fortunately, it is almost always possible to find any number of very good solutions, which may be exact solutions depending on the problem.

The fitness function serves the purpose of “natural” selection in the evolutionary system. Solutions that result in a better fitness score are more likely to reproduce and transmit their genes to the next generation. This selective reproduction is done by choosing a breeding population from the original population based on fitness score, with preference being given to those solutions with the best scores. The members of the breeding population are then paired off and children are produced through crossover and mutation.

crossover The crossover operation is a means of producing genetically related children from two parents. It is a process analogous to the recombination of DNA which forms the basis of sexual reproduction in living organisms. Reproduction as an operation distinct from crossover (by means of copying) is also possible, but does not by itself provide a search mechanism; crossover is the core of the genetic search process.

The simple crossover operation cuts the two parent strings at a randomly selected point and recombines the fragments, resulting in two new children. Each child string contains one fragment from parent A and one fragment from parent B. Multiple children may be produced from a single pair of parents through repeated crossover operations, and the combined set of all children becomes the next generation.

More complex crossover operations are possible, but the simple crossover operation described here is sufficient for most purposes.
mutation  After producing a new generation through crossover, additional variation may be introduced into the population by applying a stochastic mutation operation to some of the members. Applying a mutation operation adds noise to the search. Adding noise to a search process often improves the quality of the solution[7]. However, adding noise also slows convergence, and the highly parallel nature of genetic search makes added noise less necessary when populations are large.

6.2.2  genetic programming

Genetic programming (GP) is a powerful search technique for generating procedures that solve, or approximately solve, a programming problem. Genetic programming is a special case of genetic search in which the strings encoding the trial solution are valid sentences in some programming language. In this case, it is not a simple combination of traits that are being searched for, but rather a program (or program fragment) which that solves the problem directly.

GA vs. GP  This is an important distinction. In the genetic algorithm domain, the coded strings which represent our trial solutions can be seen as inputs to some procedure, which (if the inputs and procedure are chosen correctly) will produce the solution to the problem. These inputs can represent simple parameters, or even the addresses of subroutines or entire programs. The more complex and expressive these parameters become, the closer to GP the GA is. If GA genes are chosen to represent both functions and data, and allowed to grow to arbitrary lengths, then the GA is properly described as a GP. However, GA strings are usually fixed length representations, meaning that the search space is finite, bounded by the combinatorics of the genes. It is this upper-bound on complexity which most strongly distinguishes GA from GP.

In genetic programming, each coded string is itself a procedure, and the search occurs in the space of all possible procedures. This makes for a very flexible and powerful search process, and allows for the generation of complex procedures without the need for those procedures to be written or understood by a human.
6.2.3 valid expressions and a robust representation

Searching in the space of procedures rather than arbitrary strings complicates somewhat the crossover and mutation operations, since procedures are subject to grammar rules; For instance, simply chopping the ASCII representation of two arbitrary C expressions at a randomly chosen point and recombining will almost always produce a syntax error.

Two things are necessary in order to avoid the production of large numbers of invalid sentences in crossover and mutation operations: First, crossover and mutation must operate at the level of the syntactic structure of the language, and not its string representation. Second, a language must be chosen which has a simple, complete, robust syntax.

A GP (genetic programming) language is specified in terms of functions and terminals; functions are the operators available to the language, and terminals are the things operated on. For image filtering applications, we will assume that all terminals are real numbers representing sampled data values or user parameters. The functions available are the sampling functions described in Section 4 and all of the commonly available scalar mathematical functions, including trigonometric functions, logarithms, exponentiation, etc. Conditional expressions in the form of scalar comparisons (such as the ceiling and floor functions) may also be used.

Restricting our GP system to one type of terminal is the primary reason for not allowing a vector color notation, since allowing vector operations poses language completeness problems. Likewise, the use of Lisp for GP is particularly convenient, since Lisp syntax is simple and its parenthesized expressions have a natural tree structure[17, 18]. The crossover operation for strings has a natural extension to trees; randomly chosen subtrees of the parent expressions are exchanged to produce the child expressions. Likewise, the mutation operation works at the syntactic, symbolic level, substituting operators or arguments for syntactically valid replacements.

During crossover and mutation the number of arguments in a Lisp expression may change, or one operation may be replaced

\[
\begin{align*}
\text{Two equivalent representations: a tree and LISP expression.}
\end{align*}
\]
An early version of the GPI image filter evolution system.

with another which expects a different number of arguments. For instance, the lisp expression \((+ \ a \ b)\) in a parent expression might be replaced with expression \((\sin \ a \ b)\) in a child expression, where \(\sin\) represents the trigonometric sine function. Since \(\sin\) takes a single argument, this would ordinarily be a syntax error. This error can be avoided by defining an argument-number insensitive \(\sin\) function. Likewise, argument-type insensitivity may be added to enhance robustness if mixed data types are used. Koza uses the term *closure property* to describe a syntax for which all possible combinations of operations and operands (both in type and number) are defined[17].

6.3 evolving image filters

Karl Sims was the first to suggest the use of genetic programming for image filtering applications[24]. By adding genetic programming capabilities to the procedural image processing framework described in the previous section, it becomes possible to evolve almost any conceivable filter without the necessity of human programming.

The addition of genetic programming to the procedural design framework recasts the procedural design process as a search problem; the space of all possible image filters is very large, and the space of interesting filters (filters which the designer finds useful or beautiful, or solve a specific design problem) is usually much smaller. The goal is to find one or more of these interesting filters, and to do it with a minimum of effort on the part of the designer.

6.3.1 designer as fitness function

Were it necessary for the designer to formally specify a fitness function to match all the goals and aesthetic requirements of a particular problem, combining genetic programming with procedural design would be a very hard problem indeed. However, if the population size is manageable, then the *designer* may act as the fitness function by reviewing the results of each filter and assigning a fitness score to each filter.

There are a number of advantages to this approach. First, filters may be selected on a purely aesthetic basis, without any knowledge
or understanding of the underlying code. Second, the selection criteria can be extremely complex and, in a formal sense, ill-posed; the designer can select filters which are beautiful or have a particular “feel,” without being able to articulate what the selective criteria are. Third, this process keeps the designer “in the loop,” so that the results of the search are completely under the designer’s direct control.

The disadvantage of this approach is that population size is limited by the necessity of subjecting each filter to the review of the designer. Various interface design techniques can be used to make the selection process easier, but there are practical limits imposed by rendering time, the designer’s attention-span, and memory.

### 6.3.2 the genetic programming design process

Here, in detail, is a description of how the genetic programming design process could work when combined with the procedural image processing framework previously described. Implementation and interfacing details are described in Section 9.

1. An initial population of filters is produced. This population may be produced randomly, or may be a collection of filters already known to be “interesting” in some way or other. As has been previously described, each filter is represented by three Lisp expressions, one each for hue, saturation and value. Useful results have been obtained with population sizes as small as seven, and as large as 100.

2. The evolution interface software presents the designer with examples of the operation of each of the filters in the current population. This is done by applying each filter in the population to a common data source, which may take the form of a single image, a series of images, or a live video feed, thus allowing a side-by-side comparison between the filters and (optionally) the unfiltered source data.

3. The designer has the option of saving any or all of these filters separately for later use in a batch-processing or high-resolution rendering mode. In addition, each generation may
be saved so that evolution process can be stopped or backtrack, and restarted at any time. If the designer is satisfied with the results of the search, the process ends.

4. Based on the results of the side-by-side comparison, the designer chooses two or more image filters from the current population to be the breeding population. The designer may also specify the parameters, such as target population size, mutation rate, etc. governing the crossover and mutation process.

5. The genetic programming system (which may be an entire separate piece of software from the evolution interface) randomly pairs off the members of the breeding population and performs the crossover and mutation operations to produce a new population. This is done for each parent A and B by pairing off the hue expression of parent A with the hue expression of parent B, saturation with saturation, value with value. The children of each A, B pairing are then subject to mutation (if required), and the resulting population is handed off to the interactive genetic programming interface. The selection process begins again with step 2.

Some example images produced with evolved image filters are included in the margin here. See Section 13 for color images.
enhanced interactive genetic programming

The use of interactive genetic programming for image filter evolution has one major problem: complexity. There is a tradeoff between complexity and flexibility in the search process; the larger the population, the more flexible and powerful the evolutionary search[2]. However, the larger the search the more filters the user must examine in order to choose the members of the breeding population.

Various interface techniques can be used to manage the complexity of dealing with large filter populations, but in a conventional interactive genetic programming process the designer must evaluate every filter. This requirement is both a strength and a weakness; the designer remains in direct control of the search process, but must be directly involved in each decision. This level of involvement imposes practical limits on the size of the filter population.

If it were possible for the designer to evaluate larger populations, the flexibility of the search could be improved, and the chance of finding interesting filters in each generation would increase. One approach to achieving this is to partly automate the selection process, but how should this be done?

7.1 the APFA framework

The automated preliminary fitness assessment (APFA) framework provides an enhanced interactive genetic programming process based on preliminary fitness functions. A preliminary fitness function is a procedure which evaluates the members of a population before the user, providing a first-pass in the selection process. Preliminary fitness functions are inspired by conventional GP fitness functions and the idea of design agents[20, 12, 13]. Like an agent, a preliminary fitness function acts on behalf of the designer to review design solutions, embodying some aspect of the designers aesthetic judgment or understanding of the problem. Any number of preliminary fitness functions may be used, allowing multiple evaluation criteria.

The APFA framework is described as a progression from a relatively simple selection function chain to a more flexible system based on real-valued preliminary fitness functions. Most of the
emergent design work described in this thesis was done using the former architecture, which has proven quite useful but has a number of theoretical limitations. The extensions described in the second half of this section address those limitations, and have been partly tested in practice.

7.2 improving the search

Requiring a designer to formally specify a fitness function to match *all* the goals and aesthetic requirements of a particular problem is not realistic, since the specification of such a function is rarely well-posed. However, it is a reasonable proposition that *some* of the properties of what makes an image filter interesting could be automatically selected. Assuming this is possible, the question becomes how to do it best.

There are a number of ways this could be approached. Some desirable features for the integration of search automation are:

- It should enhance, not compromise, the power or flexibility of the search process.
- It should allow the user to automate as much or as little of the search as desired.
- It should be possible to use any combination of criteria to guide the automated search process.
- It should be a natural extension to the process and to the interface, so that it may be gracefully integrated with the rest of the system.

Possible solutions are discussed within the framework of a particular example: automatically excluding computational images from the population of evolved image filters.

7.2.1 computational images

The procedural image processing framework described in Section 4 allows the specification of filters which make no reference to the input image. Strictly speaking, these are not filters at all but computational images. For instance, the following “filter” produces the
same image regardless of input:

\[ H_d(u, v) = uv \]
\[ S_d(u, v) = v + u \]
\[ V_d(u, v) = (1 + \sin(v))/2, \]

or coded in a Lisp notation:

\[
((\ast u v) \\
(\!+ v u) \\
((/ (+ 1 (sin v)) 2)).
\]

This expression is a computational image and not a filter because it depends only on \( u \) and \( v \), and lacks hue, saturation, or value expressions.

7.2.2 the APFA selection function chain

If one wanted to exclude computational images from the population, it would not be difficult to write a selection function to identify and exclude computational images based on the filter code. This can be implemented a number of ways; for instance, an “exclude procedural images” operation can be defined within the genetic programming system, which is applied after crossover and mutation. However, excluding computational images is only one example of a useful selection function; for any given search there could be many others.

Rather than define a separate operation for each selection function (which requires a modification of the genetic programming system for each addition or subtraction of an operator) multiple selection functions can be strung together in a selection function chain. This chain can be modified by the user and applied in a single operation. In order to be allowed into the next generation, a child image filter must pass through each function in the chain.

It is important to note that the results of applying a sequence of selection functions is not order dependent. Consider two selection functions \( a \) and \( b \). In the first case, \( b \) operates alone, and rejects a subset of the filter population \( B \). In the second case, the two functions operate in sequence in the chain \( ab \); filter \( a \) rejects \( A \) and \( b \), by virtue of it being second in line, rejects \( B \setminus A \). The union of
Two parents are combined to produce four children. A selection function rejects one of the children, resulting in three image filters displayed for the user.

The two sets is \( A \cup (B \setminus A) = A \cup B \), or the union of the sets of filters rejected by both functions independently. Likewise, the same is true if the order of \( a \) and \( b \) are reversed, \( B \cup (A \setminus B) = B \cup A \). In both cases, the total set rejected is \( A \cup B \), which may be extended by induction to include the null-space (the space of rejected filters) of any number of selection functions.

The advantages of using a selection function chain include:

1. The use of a selection chain to reject uninteresting filters reduces the size of the population the user must directly evaluate, thus reducing the complexity of the search process.

2. By allowing the user to dynamically specify which filters make up the selection chain (and write new ones if necessary), the user retains (indirect) control over the initial selection process.

3. Adding or subtracting selection functions does not require a re-write of the evolution engine. Only the filter chain operation is needed, and it may be dynamically redefined during the search process if desired.

4. Compatibility with the existing process: the user may add or subtract filters from the filter chain, and a zero-element chain is equivalent to the original non-automated search process.

Experience has shown that selection filters work quite well, if they are not used too aggressively, and a small number of appropriate selection filters can significantly reduce the complexity of the interactive genetic programming process and allow the use of significantly larger search populations. However, there are a number of potential problems with this approach.

One problem is that selection functions act as binary fitness functions, which by themselves do not provide sufficient resolution for genetic search[2]. However, selection functions perform only the first pass in the selection process. Experience has shown that the combination of appropriate selection functions and the user's judgment works very well as a fitness function, producing good search results.

Another potential problem is that it is impossible to know for sure whether the selection functions are doing a good job without
inspecting the rejected population as well as the accepted one. One way to address this is by tagging filters with a boolean value to indicate their status, and passing the entire population on to be viewed by the user.

A larger problem is the non-orthogonality problem, where two selection functions using non-orthogonal criteria disagree, resulting in a high rejection rate, e.g. one selection function thinks dark images are good, another thinks dark images are bad. Since one of the goals for automating the search process is that any combination of evaluation criteria may be used, there is no way to guarantee that such a conflict will not take place. A “tug of war” between disagreeing selection functions could result in all or nearly all children failing to pass through the filter chain.

7.3 the extended APFA framework

The extended APFA framework incorporates the basic filter chain architecture, but extends it in two important ways. First, it allows the preliminary fitness functions to be real-valued rather than boolean. Second, it allows each APFA function to make an independent assessment of the population.

The extended APFA framework operates in the following way: Preliminary fitness functions, or APFA functions, assign each member of the population a fitness score. Each preliminary fitness score is a normalized value $0 < \alpha < 1$ where zero represents the best score, allowing preliminary fitness scores assigned by different APFA functions to be compared with each other. Boolean selection functions result in preliminary fitness scores of zero or one.

The result of the preliminary assessment process is an $n$-dimensional preliminary fitness vector, where $n$ is the number of APFA functions. Each dimension of the fitness vector represents the assessment of a particular function. Since the preliminary fitness scores are normalized, an overall preliminary fitness assessment can be measured as the magnitude of the fitness vector.

The preliminary fitness vector is passed back to the interactive genetic programming interface along with each image filter; no filters are rejected in the review process, so the designer still has the
opportunity to review every filter\. The advantage of the preliminary fitness assessment is that the designer can use this as a tool to provide a preliminary ranking of the new population; if the fitness functions are well chosen, the preliminary ranking is a reasonable approximation of the "true" fitness function represented by the designer, allowing most of the designer's attention to be focused on the filters with the greatest likelihood of being interesting.

The interactive genetic programming interface should provide the designer with the ability to view the new population ranked by individual fitness score or overall fitness vector magnitude. In this way, the final decision on the selection of the breeding population is still the designer's, but the designer may use automated assessment as desired.

### 7.3.1 advantages and disadvantages

The extended APFA framework has all of the advantages of a filter chain, with the following additional benefits:

1. Normalized, continuous preliminary fitness functions provide much finer discrimination than boolean selection functions.

2. The initial evaluation is a preliminary assessment, not an exclusionary process. The designer makes the final decision on all members of the population.

3. No information is lost, and the designer is free to ignore or use the preliminary fitness information as desired.

4. Multiple, non-orthogonal (or even contradictory) fitness functions may be used, with graceful results.

One problem this approach shares with the basic APFA framework is that the user (or someone else) must create the preliminary fitness functions order to take advantage of search automation. Although significant, this less of a problem than it may appear on first inspection.

---

\(^9\)The designer can always set a threshold to mimic the automatic rejection of poor scores by a filter chain, if desired.
First, some APFA functions will be useful almost all of the time, e.g. a “computational image” function which discourages computational images, or a “complexity” function which ranks filters by expression complexity. These functions represent search criteria broadly applicable to the task of image filter evolution, independent of the specific details of a particular search. Such functions can be provided with the system, and will enhance most searches.

Second, the penalty for using a non-ideal APFA function is much lower than using a non-ideal selection function. A bad filter may reject many interesting solutions, but the assessment of a bad APFA function can be easily ignored. Likewise, the results of combining many mediocre selection functions will always be no better (and frequently much worse) than any of the filters applied individually. In contrast to this, a combination of multiple mediocre selection functions may produce an overall result much better than the results of applying any one of them individually.

Third, since APFA functions operate as a heuristic aid only, the total lack of useful APFA functions can only reduce the problem to an unaugmented genetic search problem.

Fourth, it is not necessary that the APFA functions be written by a human. Although I have not implemented this approach in my research, there is a body of published work supporting the use of genetic programming to evolve agents in various contexts[4, 10], including predator-prey competitions and robot design. I outline a hypothetical structure for such a system in Section 10.8.

### 7.4 emergent design

Emergent design combines the power and flexibility of procedural design with a visual, aesthetically driven, enhanced interactive genetic programming process. Emergent design allows designers to create solutions to procedural design problems interactively and with a minimum of programming. The next section is devoted to a discussion of emergent design and its implications.
8 emergent design

Emergent design is a process, a way of approaching design problems in the computational medium. The previous sections have laid the groundwork by describing the mechanics of the emergent design process applied to the problem of developing procedural image filters. This section discusses emergent design as a theory and compares and contrasts it with other theories and processes of design.

8.1 design theory

A theory of design is a set of conceptual tools used by a designer in order to understand design problems and formulate design solutions. There are essentially three types of design theory: descriptive, normative, and prescriptive design theory. Descriptive design theory concerns itself with describing how designers solve design problems. Normative design theory concerns itself with the question of what good design is or should be. Prescriptive design theories attempt to provide a systematic process for creating good design solutions, and usually include normative elements. A subcategory of prescriptive design theory is generative theory, which focuses on specific, computational processes as a means of improving design solutions and de-emphasizes normative elements.[12].

Emergent design is a prescriptive design theory, in that it provides a procedural framework for solving design problems. More specifically, emergent design is a generative design theory; there are two reasons for this sub-categorization: First, the emergent design process depends on the use of advanced computational techniques. Second, emergent design is a process, not a standard for evaluating design. The emergent design process says nothing about what good design is, but instead provides a means for designers to achieve their design goals within a computational framework.

8.2 design tools and process

In order to understand the process of emergent design, it is useful to compare it to other design processes. On way of describing a
design process is by the types of tools used. The following categorization of design tools also provides a categorization of design process, with each successive process category defined to be a superset of the one before. The intent of this particular categorization is to be useful for the present discussion rather than to be canonical; any number of categorizations are possible.

8.2.1 non-procedural design

A non-procedural design tool is a design tool that lacks a procedural interface and produces design solutions which are not primarily represented in a procedural or programmatic form. The category of non-procedural tools is very broad, including as it does all non-digital design tools and most conventional design software. A paint-brush is a non-procedural design tool, but so is a conventional paint program.

A non-procedural design process is one in which a designer works with non-procedural tools to solve design problems. This definition includes the sophisticated use of design software, if that software does not provide the designer with a procedural design interface. The non-procedural design process is used by the vast majority of practicing designers.

8.2.2 procedural design

A procedural design tool is a design tool that provides a procedural, or programming interface for creating design solutions. The results of using a procedural tool is a procedural design solution. A procedural design solution is a design solution or a portion of a design solution which is best described or most naturally represented by a program or procedure.

The procedural image processing framework described in Section 5 is an example of a procedural design tool, and a procedural image filter created with this system as part of a design process is an example of a procedural design solution.

A procedural design process is one in which the designer works with procedural design tools to solve design problems; the use of non-procedural tools may also be part of the process. Procedural design tools allow a design process to be described in procedural or programming terms. The advantage of a procedural design
process over a non-procedural design process is the flexibility and power provided by procedural design tools. The disadvantage is the difficulty presented by working with a procedural or programming interface.

8.2.3 participatory procedural design

A participatory procedural design tool is a design tool which allows for the creation of procedural design solutions without the use of a programming interface. Participatory procedural design tools are so-named because they allow a wider class of designers to participate in the creation of procedural design solutions. An example of a participatory procedural design tool is the system for evolving computational images described by Karl Sims[24], or the image filter evolution framework described in Section 6.

A participatory procedural design process is one in which the designer works with participatory procedural design tools. Ordinary procedural design tools and non-procedural tools may also be used. The participatory procedural design process provides the designer with the power of the procedural design process. To the extent that participatory procedural vs. non-participatory procedural design tools are used, the participatory procedural design process requires less technical skill than the procedural design process. A design process involving the use of conventional interactive genetic programming is an example of a participatory procedural design process.

8.2.4 emergent design

An emergent design tool is a particular type of participatory design tool which employs an enhanced interactive genetic programming system, as described in Section 7. An emergent design process is a specific type of participatory procedural design process in which the designer works with emergent design tools. Other types of tools may also be used. The emergent design methodology and its benefits are discussed in more detail below.
8.3 emergent design and the model of dynamic design

The preliminary fitness functions of the APFA framework serve an agent-like role in guiding the search process; the Model of Dynamic Design proposed by Suguru Ishizaki[12, 13] uses agents as dynamic composition elements. In Ishizaki’s Ph.D. thesis, he proposes a model for creating dynamic design solutions through the metaphor of performance. In this model, the designer takes on a role analogous to the director of an improvisational dance troupe, and “trains” a group of design agents to perform the dynamic composition. This group of agents, the design system, is expected to perform independently of the supervision of the designer.

In emergent design, the preliminary fitness functions are collaborators or helpers for a designer actively engaged in solving design problems. The design system is an interactive tool for the designer, not for the end user. The difference is subtle, but important; emergent design is a process for extending the capabilities of the designer, the Model of Dynamic Design is a methodology for creating dynamic compositional performances which function independently of designer.

8.3.1 justifying the emergent design processes

Previous sections of this document have taken a ground-up approach to describing the emergent design process, beginning with a specific problem (image processing) and a specific set of tools; grounding the discussion in specific examples provides the easiest path to understanding what emergent design is and why it might be useful. However, this discussion may not answer all of the reader’s questions about the emergent design process.

One of the most important questions is why? Why bother going through all of the trouble of implementing this admittedly complex computational framework when designers have been working successfully with much simpler tools? And if designers are to work with sophisticated computational tools, why not use sophisticated, well-established conventional design software?

The answers to both of these questions are related, and their relationship hinges on the power of the computer. The computer is an amazingly powerful tool. It is a tool like no other that has existed in human history. The great promise of the computer is that
for the first time we have a tool that can truly extend the human mind as well as the human hands and voice. It is a qualitative, precise statement that what distinguishes a computational process from a simpler process is that the results of the former can not, in general, be known in advance. This is another way of restating the halting problem[9, 26], but in my mind it strikes at the essence of what makes computation special: computation can show us things that cannot be seen or understood in any other way. Computation can show us the unexpected, and at the same time every part of the process is completely deterministic, repeatable, and knowable.

Simple, non-computational design tools can surprise the designer, especially when the designer is first learning to use the tools. However, once a certain level of mastery is reached, the surprises become fewer and farther between. This is also true of sophisticated conventional design software which uses non-computational tools as an interface metaphor. Surprise is very important, because it is the surprises, the mistakes that provide new insight as much as any reflective process.

A procedural design tool is a tool for constructing a computational process. The results of any non-trivial procedure cannot be known in advance. This is not the result of mistakes, or a limitation in the designer’s understanding, but a fundamental property of the system. Procedural tools have within them an almost limitless capability for surprise, which comes at the cost of requiring a high level of technical skill for their use.

This is the justification for using participatory procedural tools. If these tools can be made useful, they hold the promise of providing even greater variation and surprise than procedural tools, with little or no programming knowledge required in their use. Participatory procedural design tools provide an even greater capacity for surprise than conventional procedural tools because through the participatory procedural design process it is possible to discover procedural design solutions which the designer would never have considered writing, even if the designer were a skilled programmer.

The challenge is to create a framework for participatory procedural design powerful enough to produce any conceivable procedural design solution, yet simple and straightforward to use. I believe that emergent design meets this challenge.
9 implementation

This section discusses the implementation of the evolution++ emergent design system, beginning with a discussion of the implementation of the Sol programming language as a basis for genetic programming.

9.1 a procedural design language

Working with procedural design tools is essentially a programming task, which raises the question of what language to use. Special purpose languages may be appropriate for specific design applications, but a full range of problems can only be addressed by a general purpose programming language. One of the central principles of computer science is the Church-Turing thesis[9, 26], which states that all general-purpose programming languages are fundamentally equivalent, i.e. all programs in one general purpose language can be translated into any other general-purpose language. Just because this is possible does not make it easy, and as a practical matter some languages are much better for certain tasks than others.

Early in my work in the Aesthetics and Computation Group, I became interested in the idea of an ideal procedural design language. The central question this raises is the definition of a such a language: What features would an ideal procedural design language have? I decided that the best, and perhaps the only way to answer this question was in the framework of my own experience and needs as a designer.

9.1.1 the features of an ideal language

The structure and syntax of the language should reflect my aesthetic values, including elegance, beauty, functionality, and clarity. Both as a designer and programmer, I look to the world of physics and nature for inspiration in solving problems. Natural processes are bottom-up phenomena, where higher level structure emerges from the interaction of simpler, lower-level elements. Bottom-up processes are best described in terms of parallel operations, so the language should be well suited to expressing parallel as well as sequential computation.
The ideal language would be useful both as a design systems development language and a procedural design interface language. As a design-systems development language it should compile to a relatively efficient representation, so that performance does not suffer in large or computationally demanding tasks. As a procedural design interface language, the ideal language would be capable of run-time code generation and manipulation, a necessary feature for genetic programming applications. The language should be highly portable, well supported, and useful for creating networked and/or world wide web applications.

9.1.2 existing languages

Based on these criteria, I evaluated a number of existing languages, including C++, Scheme Lisp, and Java:

C++ C++ is my systems-programming language of choice. Its object-oriented features, practical C-derived syntax, and generally high performance make it a good choice for many tasks. C++ benefits from being well supported and fairly portable at the source-code level. A number of excellent tools are available for C++ development, including the freely available, open-source Gnu C++ compiler and associated libraries. Support for networked client/server programming is available through standard libraries, but Java-like web applet support is not available.

C++ suffers aesthetically, being expedient rather than elegant. Parallelism at the thread level is possible in C++, but not directly supported by the language specification. Likewise, there is no easy way to do run-time code generation and modification. Overall it is a very useful language, but not ideal for all procedural design applications.

Java Java, a language developed by Sun Microsystems\textsuperscript{10} is a popular general-purpose programming language derived from C++. Java's simplified C++ syntax is easier to learn and use than C++,

\textsuperscript{10}Information on Java, including the language specification, can be found at http://java.sun.com
and unlike C++\textsuperscript{11} Java provides support for parallel operations in the form of a native threading syntax. Java source code compiles to a byte-code representation which executes on a standardized, well supported virtual machine. For this reason Java source and compiled code is highly (though not perfectly) portable. In addition, Java provides a set of standardized packages and libraries which make some tasks, like creating web applets, network programming or simple GUI development, relatively easy. Java virtual machines can be embedded in other applications, making Java suitable as an extension language.

Java’s syntax is considerably cleaner than C++, but is still not particularly elegant when compared to a language like Scheme. Java’s high portability comes at the cost of performance, which is roughly an order of magnitude slower (or worse) than well-compiled native code on most platforms. Java’s native GUI development and graphics library, the AWT, makes simple GUI development easy but sophisticated graphics difficult. Java lacks a simple mechanism for for run-time code generation and modification, though in principle this is easier to do in Java than in C++. Like C++, Java is useful for many applications, but is not an ideal procedural design language.

\textbf{Scheme} The Scheme dialect of Lisp\textsuperscript{15} is an extremely pure, elegant language. Lisp in general, and Scheme in particular, are well suited to run-time code generation and manipulation. Scheme’s powerful recursive syntax and simplicity make it the language of choice for many computer-science research applications, including artificial intelligence. Scheme is somewhat portable at the source-code level, and can be used both as a software development and an interface or extension language.

Unfortunately, Scheme is not very practical for many common programming tasks, largely because of its traditionally poor performance. In addition, support for networking and graphics is not standardized, and Scheme lacks a mechanism for strong, explicit typing. Scheme has no native syntax for parallel computation, and its portability is hurt by significant variations between implementations, despite the attempts at standardization put forward in the

\textsuperscript{11}As I mention above, it is possible to write threaded applications in C++, but threading is not part of the C++ language specification.
IEEE standard[11] and R5RS report on the language[15]. There are portable Scheme runtime environments such as MIT Scheme12, but these are not widely used outside of academic computer science settings. Poor performance and lack of parallelism are the two major reasons why Scheme is not an ideal procedural design language.

other languages  I investigated dozens of other languages in my search for an ideal general purpose procedural design language. Some, like Mitchel Resnick’s Star Logo[22] or the Swarm system developed by Nelson Minar and others at the Santa Fe Institute[21], were very interesting, but too specialized. Others, like SETL[23], were obscure, dated, or otherwise difficult to work with. Many of these languages have interesting features, but it is impossible to review them all here. Of all of the existing languages I investigated, the three mentioned above (C++, Java, and Scheme) were the ones I found most useful, though none were ideal for all applications.

9.2 Sol

My review of programming languages lead me to believe that it might be possible to combine the better features of several languages to create a new language closer to the ideal. Of all of the languages I reviewed, Scheme seemed the closest, and was a natural choice for a starting point.

At the time I began work on this project, it seemed reasonable to assume that Java virtual machine performance would improve considerably through just-in-time compiler technology and improved garbage collection. Targeting the Java virtual machine promised great benefits in cross-platform compatibility, and interoperability with the growing collection of useful Java libraries and APIs. In addition, targeting the JVM would allow the creation of web applets as well as applications, and tie into Java’s extensive network programming features.

For these reasons, I decided that a more practical version of Scheme with support for parallel operations running on the Java virtual machine would be the goal of the project. Several Scheme-to-JVM (Java virtual machine) compilers existed or were in the

12http://www-swiss.ai.mit.edu/scheme-home.html
early stages of development, and it appeared to be technically feasible to extend one of these to add the desired language features of parallelism and strong typing.

Because of the relative maturity of the effort, I chose to use the Kawa compiler as my starting point. Kawa is an open-source Scheme-to-JVM compiler being developed by Per Brothener[3]. By introducing sets and set operations into the language, I added the features of parallelism and strong typing. The result is a language I call Sol, which stands for Set Operations Language[5]. For a more detailed description of Sol and its features, see Section 12.

The Sol language achieves most of what I hoped for, but the current implementation of the language has a few problems, primarily in the areas of performance and portability. This is discussed in detail in Section 10.1.

9.3 genetic programming in Sol

As a dialect of Scheme, Sol provides a strong basis for genetic programming. Much of this it inherits from Scheme itself, since Lisp syntax is particularly appropriate for genetic programming applications[17, 18, 2]. However, Sol’s set and set-operation notation provide additional benefits for this highly parallel task.

The basic genetic programming operations of fitness evaluation, selection, reproduction through crossover, and mutation are all well expressed in terms of set operations. For instance, let the set of lists be denoted \( L \), and the set of lists of procedures denoted \( L_p \in L \). The set of lists of real numbers \( 0 \leq x < 1 \) will be represented as \( L_n \in L \).

Consider a population of gray-scale image filters, in which each member is a single procedure. Because we need to keep track of (possibly multiple) fitness evaluations, each member will also have associated with it an initially empty list for storing these evaluations. The result of this association is a compound object composed of a procedure and a list of numbers, or \( L_p \times L_n \).

The population as a whole, \( P \), is composed of individual members, \( p \in P \), \( P \subseteq L_p \times L_n \). A generation \( G_i \) represents the subset of the population produced at generation \( i \), where the initial generation is \( G_0 \). The population as a whole is the union of all generations, or \( P = G_0 \cup G_1 \cup \ldots \cup G_n \).
Having defined this notation, we can talk about set operators which produce a new set from an existing set. In the case of genetic programming operations, each operator \( f: L_P \times L_n \rightarrow L_P \times L_n \) has the property of producing a set of valid population members when operating on a set of valid population members.

The fitness set operator \( e \) produces a new set identical to its argument set, except that the (presumably empty) fitness list of each element is replaced with a new list of fitness evaluations. The selection operator \( s \) stochastically selects a subset of its argument set based on the fitness evaluations of each element. (In interactive genetic programming, \( s \) is the user’s judgement\(^{13}\).) The crossover set operator \( c \) stochastically pairs off the members of its argument set and applies the crossover operation to produce a new set of children. The mutation set operation \( m \) stochastically applies the mutation operation to its argument set, resulting in a new set containing mutated members. Thus, the production of a new generation from the current generation may be written as:

\[
G_{i+1} = m(c(s(e(G_i))))
\]

9.3.1 operators in Sol

When programming in Sol, describing the mechanics of genetic programming in this way is not merely a notational convenience. For instance, here is the actual implementation of the mutation set operator used in the evolution++ genetic programming system:

```sol
(operator ;; An operator...
() ;; taking no additional ;; arguments.

(ordered ;; Return an ordered set...
(list ;; composed of a single list ;; of three procedures and a ;; fitness vector:
   (uv-mutate (car this-element))
```

\(^{13}\)Or a combination of the user’s judgement and an automated preliminary fitness assessment system.
(uv-mutate (cadr this-element))

;; Mutate the third procedure.
(uv-mutate (caddr this-element))

;; The unmodified fitness vector.
(caddr this-element)

The internal details are less important than the use of the operator syntax, which allows for the specification of n-ary set operators in Sol. Operators are applied in parallel to all the members of a set, resulting in a new set. Sets, operators, member functions and data can be grouped into categories, which provide many of the features of data and procedure encapsulation provided by classes in a conventional object oriented programming language.

The evolution++ genetic programming server is implemented as a Sol category in which all of the basic genetic programming operations are implemented as operators, or function wrappers around operators. This syntax is very clean and allows the parallel nature of these operations to be explicitly stated\(^\text{14}\).

### 9.4 GPI and evolution++

The evolution++ system is a hybrid Sol/C++ implementation of an interactive, APFA enhanced interactive genetic programming system for the evolution of real-time video filters. The evolution++ system grew out of an earlier, non-real-time filter evolution system for still images or image sequences, which I call GPI.

#### 9.4.1 GPI

The implementation of GPI is interesting for several reasons. First, it is a Sol applet which uses the Java AWT and a number of other Java packages, demonstrating the relative ease of interfacing Sol and Java code. Second, it uses a Sol category representation to implement the genetic programming process, as was discussed above.

\(^{14}\)For more information on sets, operators, categories or other features of Sol see Section 12.
GPI is based on the procedural image processing framework described in Section 5 and the genetic programming system described in Section 6, but does not employ APFA functions. (Later versions of GPI employ a simple procedural image filter, as described in Section 7.) GPI provides a very simple visual interface, in which all of the members of a generation (typically fewer than ten filters) are shown as small rendered “thumbnail” images. Images are imported and exported from the system in JPEG format, and all filtering is done in HSV space.

By clicking on a thumbnail with the mouse, the designer is able to view and edit the filter code directly, or render a large image or image sequence.

The interactive genetic programming interface is simple, and geared to the small generation sizes typically used with this system. The user is allowed to pick two filters from the current generation to be parents of the next. The currently selected parents are displayed at the bottom of the main panel as larger rendered thumbnails. Sliders on the main applet panel provide access to parameters governing the crossover and mutation process.

GPI allows the user to work with a single source image or sequence of source images, linearly interpolating between pixel samples, and in the case of image sequences, linearly interpolating between frames. The result is a continuous image space, which can be sampled in complex ways.

This three-dimensional image-space sampling leads to interesting effects, particularly when the sequence of source images represent a progression in time. I call this effect temporal warping, and discuss it further in Section 10.

Because GPI allows arbitrary sampling in the three-dimensional image space, and because each expression (hue, saturation, and value) can sample differently, it is often difficult to understand where a particular filter samples the image space. In order to explicate this process, I created a pure-Java visualization application which produces a three-dimensional visualization of the sampling process used to create a sampled image.

By sketching on the image, the user draws on a 3D visualization of the hue, saturation, and value sampling surfaces used to create the image. In this way, the user can see exactly where the filter is sampling the space to create the resulting image. This vi-
sualization is useful both in explaining image-space sampling in
general and showing the operation of a particular filter.

9.4.2 GPI and APFA

Later versions of the GPI system employ a hard-coded selection
function to exclude computational images. This proved very use-
ful, and laid the ground-work for the more sophisticated APFA
framework implemented in the evolution++ system.

9.5 evolution++

The evolution++ system grew out of my frustration with the lim-
itations of the GPI system, which are discussed in Section 10. The
main problem is performance: Image filtering requires at least one
computation for each pixel in the output image. Even on a rel-
atively fast machine, the Sol/Java GPI implementation (both Sol
and Java compile to JVM byte code) is frustratingly slow. The
rendering time for filtering large images approaches an hour for
some images and filters.

For this reason I decided to construct a new system with a
C++/OpenGl front end, which would run on our SGI Octane and
O2 computers. The C++ code provides image filter rendering at
real-time or near-real-time speeds, resulting in an interactive video
filtering system which employs the emergent design methodol-
gy for evolving procedural video and image filters. Creating the
 evolution++ system posed three major technical challenges:

1. Interfacing the separate Sol and C++ programs.

2. Translating Sol expressions for image filters into C++.

3. Performing run-time compilation of the resulting C++ fil-
ter code and dynamically linking it against the running C++
front-end.

9.5.1 interfacing Sol and C++

The first problem, that of interfacing the Sol and C++ programs,
is solved through a simple client-server architecture. My origi-
nal plan was to create a Sol genetic programming server which
would talk to the world through TCP/IP sockets. Such a framework would provide many advantages, including the ability to support multiple clients simultaneously and to support multiple types of clients, including web-based clients. The disadvantage is that multi-threaded server code must be very robust, as an experiment with creating a web-based distributed Pi computation client/server system demonstrated.

Although I had a Java-based multi-threaded generic server framework in place, in the interest of time I decided to build a single-threaded genetic programming server using Unix named pipes rather than sockets for client/server communications. This system is simpler, provides tighter integration between client and server, but requires that both client and server be on the same machine, or on machines which both have access to the same networked file system.

9.5.2 translating Sol filters into C++

Translating the Sol image filter expressions into C++ is a straightforward, though somewhat complicated process. Translating the individual filters is relatively simple, almost a text-book exercise in translating a prefix notation (Sol) to infix (C++) syntax. The complexity arises from the strategy of wrapping C++ classes around groups of filters; this provides a number of classic object-oriented programming advantages, including data encapsulation, name-space segmentation, method inheritance, and simplification of the dynamic linking process by reducing the number of public library symbols. The added complexity comes in part from C++ restrictions on casting class member function pointers, which is a messy subject[27].

Translating the image filters is handled by the Sol genetic programming server, which performs the translation as the last step of the process of creating a new generation of image filters. For example, here is the Sol code for an actual evolution++ procedural image filter produced during a ten generation run:

```cpp
;; Filter generation1010, filter 1

'(
  (* (hue* (min* 50.0 u)) (- (* 50.0 u))) ;; hue
```
Note that the filters are grouped by generation, and that the Sol filter representation includes heredity and fitness vector information in a comparatively compact representation.

Here is the same filter, now translated into C++.

```c++
/* Filter generation1007::filter1 */
/*================================----*/
/* generation1007::filter1 hue */
float generation1007::filter1hue(float u, float v, float time) {
    return (hue(((50.0 < u) ? 50.0 : u), 0) * (50.0 * u));
}

/* generation1007::filter1 saturation */
float generation1007::filter1saturation(float u, float v, float time) {
    return ((v < (v * 0.5 * fsin((6.28 * u))))
     ? v : (v * 0.5 * fsin((6.28 * u))));
}

/* generation1007::filter1 value */
float generation1007::filter1value(float u, float v, float time) {
    return (fcos((value(v, u) * value(u, v)) - u);
}
```

71
char *generation1010::string1hue
   = "(* (hue* (min* 50.0 u)) (- (* 50.0 u)))";

/* generation1010::string1 saturation */
char *generation1010::string1saturation
   = "(min* v (* v 0.5 (sin* (* 6.28 u))))";

/* generation1010::string1 value */
char *generation1010::string1value
   = "(- (cos* (* (value* v u) (value* u v))) u)";

/* generation1010::string1 parentA */
char *generation1010::string1parentA
   = "generation1007";

/* generation1010::string1 parentB */
char *generation1010::string1parentB
   = "generation1009";

/* generation1010::string1 indexA */
int generation1010::string1indexA = 2;

/* generation1010::string1 indexB */
int generation1010::string1indexB = 1;

/* generation1010::fitnessVector1 */
float *generation1010::fitnessVector1 = NULL;

/* generation1010::fitnessVector1Length */
int generation1010::fitnessVector1Length = 0;

This representation is comparatively verbose at the source-code level.

**the C++ representation and introns** Solutions produced through genetic programming often contain *introns*, or code sequences which do not affect the results of the computation. Introns are a natural product of the genetic programming process, and may actually play a useful role in the evolution of solutions, at least in the early stages of the process[2].

The translated C++ filter code includes the original Sol filter expressions as strings, since this information must be preserved for future genetic programming use and cannot be recovered from the compiled C++ hue, saturation, and value procedures. This is not simply because run-time disassembly of a compiled C++ procedure and translation back into Sol would be hard, but also because the original Sol expression may contain introns in the form of additional, unused arguments which are lost when translated into C++.

**9.5.3 run-time compilation and dynamic linking**

As has been previously discussed, the genetic programming server produces C++ code which must be compiled and linked against the C++ front end program. One way to accomplish this would be to stop the program, compile the new filter code, link the program against the new filter code, and restart the program. Although stopping, compiling, and restarting could be automated through the use of a shell script or some other program, this type of interruption (which would last several seconds) would seriously detract from the interactivity of the genetic programming process.

Instead, the new C++ image filter code is compiled and linked against the running C++ front end program. Compiling the C++ code in a sub-process while the front end is running is straightforward, but dynamic linking is more difficult. Fortunately, the dlopen dynamic shared-object linking interface makes this possible under SGI IRIX, and a number of other Unix operating systems. The dlopen interface allows symbols to be resolved from
a shared-object library at run-time, if the name of the library, the symbol and the type of object the symbol refers to are known.

One complication is C++ name mangling, in which the name of a compiled C++ function is “mangled” to encode the type and number of arguments expected by the function\[27\]. Without knowing the mangling strategy of a particular C++ compiler, extracting functions from a DSO (dynamic shared-object library) becomes difficult. This problem was solved by building the DSO once by hand and then dumping out the library symbols, which included the mangled function names. Another solution may be to declare a function “extern c”, which prevents the compiler from mangling the function name.

9.6 the evolution++ video processing framework

The evolution++ video processing framework is implemented on top of the ACU\[15\] library video functions, which I hacked somewhat for performance. This library provides a thin layer of functionality over the native IRIX vl video library. The evolution++ VideoSource class converts the raw RGBA video frame provided by the SGI video hardware into a floating-point HSV representation for use by subclasses of VideoFilter. The VideoSource class also provides methods to convert the HSV floating-point results of image filtering back to RGBA for display. A subclass of VideoSource called ImageVideoSource produces HSV image data from a sequence of image files instead of the video hardware, for use in off-line rendering applications.

Each generation of Sol image filters produced through the interactive evolution interface is translated into a subclass of SolFilter, which is in turn a subclass of VideoFilter. Each image filter in a SolFilter instance may be applied separately to the HSV image provided by a VideoSource instance, or all may be applied simultaneously in a “tiled” mode for side-by-side comparison. The image rendered by a VideoFilter may be any resolution, but is typically no larger than the resolution of the image provided by the VideoSource. Any number of SolFilter instances may be used, although performance issues impose a practical limit of three

\[15\]ACU is a utility library developed in the ACG which extends OpenGl and Glut. It provides, among other things, an interface to the IRIX vl video library.
or four (depending on resolution) for interactive applications.

9.7 the evolution++ user interface

The evolution++ user interface (which runs in full-screen mode) displays a maximum of four generations of image filters (four SolFilter instances) simultaneously. The designer interacts with the system through the mouse and keyboard. By clicking with the mouse, the designer designates filters to be parents of the next generation, to be “hot” filters saved out separately for later use.

The current generation of filters is rendered the largest, and located in the upper right of the screen. Three previous generations are visible at the left, with the most recent at the top. Below the current generation, either the video source itself or a text-based command monitor may be shown.

Beside and above each SolFilter rendering are symbols which indicate the index of the currently selected filter, its status as a parent or hot, and the magnitude of its preliminary fitness vector (if any). If APFA functions are used, the index of the filters reflects the overall preliminary ranking by the APFA functions. By mousing over a SolFilter and hitting the t key, the user may toggle “tiled” mode display. In tiled mode, the current filter is selected by positioning the mouse over the tile of interest. Hitting the t key again puts the SolFilter back into single-filter display mode, now showing the filter associated with the previously selected tile.

It is often useful to see exactly which filters are the parents of some other filter; the evolution++ user interface provides a heredity mode (selected by the h key) to visualize this information. By positioning the pointer over the filter of interest, all other filters are hidden except for the currently selected filter and that filter’s parents.

Parents of the next generation may be chosen from any of the four generations visible, but all generations produced in a run are saved on disk. At any time a run may be exited by pressing the escape key. Once exited, a run may be restarted at any generation, from the current (last) to the first. This means that no information is lost in the evolution process, and all filters produced are available for later use. In addition, the designer may back up the evolution process and restart at some previous generation if de-
sired.

Once parents are selected, the designer produces the next generation by pressing the ! key, which causes the selected filters to be sent to the genetic programming server. The filters resulting from the crossover, mutation, and preliminary fitness assessment are compiled, and the new library is dynamically linked. (The details of this process, including the dialog with the genetic programming server, are displayed on the command monitor.) The new generation becomes the current, and the other generations shift down the display.

Filters which the designer designates as hot are saved out separately, and can be combined into collections. These collections may be used in a batch processing mode, in which the designer scripts holds, cuts, and cross-dissolves between filters using a simple scripting language. Batch processing mode is primarily intended to be used in off-line rendering, where an ImageVideoSource is used to provide source image data from a sequence of image files.

9.8 the APFA framework

The APFA framework is implemented as a collection of preliminary fitness functions, which are applied by the genetic programming server after crossover and mutation. The user selects which APFA functions to apply by adding them to a list of Sol operators in the Agents.scm file.

Here is an actual example of a selection function, implemented in the Sol programming language. This selection function is a combination of four simpler selection functions which count references to the symbols u and v in the filter expressions. The purpose of this function is to ensure that both u and v are referenced. This is important, because otherwise the two-dimensional quality of the source image is lost.

The u-and-v selection function and the no-comp-images preliminary fitness function (see below) provide an effective combination to reject computational images and filters which don’t sample the entire source image. These two APFA functions, occasionally combined with a simple complexity measure function, have proved very useful in evolving interesting and beautiful image filters.
(define u-and-v
  (operator ;; a Sol operator
    ((uv-hue boolean) ;; type-checking argument notation.
      (uv-sat boolean) ;; This operator expects four booleans as additional
      (uv-val boolean) ;; arguments (this-set, this-element, and
      (uv-any boolean) ;; this-index are the implicit operator arguments).
    )
  )
(display "u-and-v called") (newline) ;; debugging info
(let ((hue (car this-element)) ;; Separate out the
  (sat (cadr this-element)) ;; various bits of
  (val (caddr this-element)) ;; the image filter.
  (u-list 'u)
  (v-list 'v)
)
(let ((hue-u-count
  (count-symbols hue u-list)) ;; Count-symbols does
  (hue-v-count
    (count-symbols hue v-list))
  (sat-u-count
    (count-symbols sat u-list))
  (sat-v-count
    (count-symbols sat v-list))
  (val-u-count
    (count-symbols val u-list))
  (val-v-count
    (count-symbols val v-list))
  )
(let ((hue-case ;; check the cases
  (and (> hue-u-count 0) (> hue-v-count 0)))
  (sat-case
    (and (> sat-u-count 0) (> sat-v-count 0)))
  (val-case
    (and (> val-u-count 0) (> val-v-count 0)))
  )
)
(if (and ;; accept or reject.
Here is the code for an implementation of the no-comp-images function, implemented as a real-valued preliminary fitness function for the extended APFA framework. Note that a fitness score is explicitly used, even though in this case it’s either 0 or 1 − ϵ.

(define no-comp-images
  (operator ; A Sol operator...
    ((min-hue integer) ;; ...taking four integers
      (min-sat integer) ;; as arguments.
      (min-val integer)
      (min-total integer)
    )
  )
  (display "no-comp-images called") (newline)
  (let ((hue (car this-element))
    (sat (cadr this-element))
    (val (caddr this-element))
    (heredity (cadddr this-element))
    (fitness (cadddr (cdr this-element)))
    (symbols ; This is the list of sampling
      '(sample* ;; functions we want to count.
        hue* saturation* value*
(let ((hue-count (count-symbols hue symbols))
  (sat-count (count-symbols sat symbols))
  (val-count (count-symbols val symbols)))
  
  (if (and ;; do the comparisons
    (>= hue-count min-hue)
    (>= sat-count min-sat)
    (>= val-count min-val)
    (>= (+ hue-count sat-count val-count) min-total)
    
    (ordered ;; Return a good fitness score
      (list ;; (zero is best) as an ordered
        hue ;; set containing one list (the
        sat ;; image filter with a zero
        val ;; appended to its fitness vector).
        heredity
        (cons 0 fitness)
      )
    )
  )
  
  (begin
    (display "no-proc-images be dising ")
    (display this-element)(newline)
    (ordered
      (list ;; Return a bad fitness score.
        hue
        sat
        val
        heredity
        (cons 0.9999 fitness) ;; 0.999 = 1.0 - epsilon
      )
    )
  ))
))
10 results and conclusions

This section discusses the results of my research, draws conclusions from those results, and discusses future research directions. The enhanced genetic programming results are the most important, so a brief summary is provided here.

I found interactive genetic programming, with and without APFA enhancements, to be a useful design tool. Based on my research I believe that there are four important factors for making interactive GP work well in image processing applications:

1. The use of the HSV color space. As I explain in Section 5.4.4, working in HSV space greatly reduces the complexity of useful filtering operations. The result is that useful filters are simpler, making them easier to find with interactive genetic programming. Both GPI and evolution++ use HSV filtering.

2. Using carefully chosen initial populations for small generation-size searches. Useful results may be obtained with generation sizes as small as seven, if the initial population is chosen well. This is demonstrated by the GPI tool, which allows the specification of the initial population and is performance-limited to generation sizes of fewer than ten filters.

3. The use of a high-performance filter rendering system for large generation sizes. It is impossible to do useful interactive genetic programming if the user is forced to wait minutes for each generation. By translating the image filters to C++ and performing run-time compilation and dynamic linking, the evolution++ system allows for the rendering of hundreds of image filter “thumbnails” at interactive rates. This is key to achieving the breadth of search demonstrated by the GoVaul project discussed below.

4. Providing a preliminary fitness framework, even a simple one, can significantly improve the process. For instance, from an initial breeding population containing no computational images, fully one-third of the children can be expected to be computational images that are recognizable by
a simple selection function. By eliminating these from consideration, one APFA function can reduce the complexity of the designer’s interaction by a third.

I believe the results of this research support the hypothesis that participatory design in general and emergent design in particular can be effective for creating visual design, at least in an image manipulation context. The rest of this section is devoted to a detailed discussion of these results.

10.1 Sol

The Sol programming language has proven itself a useful procedural design tool. The language provides all of the features outlined in Section 9.1 for an ideal procedural design language, except in the area of performance; the current Sol implementation has performance problems, which in part may be due to the structure of the language itself. (I will say more about Sol’s performance problems shortly.) As a practical matter, it is important to distinguish between the language and the implementation, since the current implementation is a prototype and is by no means optimal.

10.1.1 aesthetics

Sol inherits the elegance of the Scheme programming language, and succeeds in maintaining much of Scheme’s syntactic purity. Sol sets and categories fit naturally into Scheme’s syntax, and the integration of the literate programming system allows for a very clean way of commenting and documenting code.

The only real source of ugliness in the current language implementation is the limited use of the C preprocessor to support multiple underlying Scheme implementations (MIT Scheme and Kawa), and in compiling Sol web applets. Fortunately, this is an implementation issue and not a necessary language feature; the use of the C preprocessor could be obviated by creating a fully integrated Sol compiler which does not rely on an underlying Scheme compiler.
10.1.2 parallelism

Sol’s use of set operations as syntax for parallel computation is clean, unambiguous, and concise. The current implementation does not fully support this parallelism (data parallel operations are executed sequentially), but the notation is still useful for designating inherently parallel operations.

10.1.3 application and interface language

As a system for creating applications, the Sol language system works fairly well. The integrated literate programming system (inspired by Donald Knuth’s Web[16]) allows for the production of typeset \TeX\ documentation and compiled code from the same source file. The compilation process works reliably, and the Sol/Java language interface features allow Java libraries to be used as though they were native Sol categories.

The ideal procedural design language would be useful both as an application development language and as an interface/extension language; Sol’s use in the GPI interactive genetic programming system demonstrate it’s usefulness in this regard. The GPI system is written in both Sol and Java, with Sol used as the main application development language and as the interface language. Java is used in a supporting role where it is necessary to subclass AWT widgets or in some of the computationally-intensive image processing tasks.

The combination of Sol as the main application language and the extension/interface language is particularly powerful, since this allows for a seamless integration between application and interface. The interface language code is compiled on-the-fly to the same internal representation as the main application code, and provides access to any desired application resources. Once interface code is evaluated, it functions just as though it were part of the original application program. Using this framework interface code can call any globally defined procedure, interact with the user directly by creating new GUI elements, or access operating system resources (files and other programs) directly. For instance, the time-parameter used in this GPI image filter is not passed as a parameter to the filter procedure, but a global symbol in the GPI application itself which is referenced directly by the filter code.

```
((hue-space* u v (-
    time-parameter (* 0.33 u)))
 (saturation* u v (*
    time-parameter 10.0)))
 (value-space* u v (-
    time-parameter (* 0.33 u))))
```

GPI image filter code.
10.1.4 compilation and execution efficiency

The ideal procedural design language would compile quickly and execute efficiently. As a prototype, the current implementation of the Sol language system functions fairly well, but compilation and execution efficiency are less than ideal.

Compiled Sol code generally executes slower than equivalent pure Java code, often by a factor of two or more. Even worse, compilation times can be very long when compared to the compilation time for an equivalent number of lines of C++ or Java.

There are a number of reasons why the performance of the current implementation is poor, including performance-limiting features of the language, the use of the Java Virtual Machine as the target platform, and implementation time constraints.

performance-limiting language features

Compilation and execution performance are properties of the language implementation, but there are features of the Sol language specification which make efficient compilation and execution more difficult to implement. Two language features in particular pose performance difficulties: run-time code evaluation and garbage collection.

Languages like Sol and Scheme which allow for run-time evaluation of code, as in the use of the eval statement, must provide a means of compiling code on-the-fly in the run-time language system. One implication of this is that the run-time system must include a compiler, which increases the size and complexity of the resulting application code. It also makes efficient compilation more difficult, since the compiler can make fewer assumptions about the definition of core symbols and application structure. The prototype Sol implementation relies heavily on the Kawa Scheme-to-Java-VM compiler, and the performance of Sol depends in large measure on the efficiency of this compiler.

Garbage-collected languages like Sol and Java must devote a non-trivial amount of time and resources to memory management. The Sol language system compiles Sol source code to Java virtual machine byte-code; modern JVM’s perform run-time compilation of Java byte-code to native machine code, but the execution performance of both Sol and Java is still limited by the garbage collection efficiency of the JVM.
implementation time constraints  

At each step in the development of the Sol language system, I was forced to make tradeoffs between developing the language quickly for use in this research and creating an optimal implementation. In some ways dealing with the tight time constraints was positive, in that it forced me to use existing tools in creative ways and to resist the temptation to re-implement the wheel. In other ways it was negative; the available tools were not always optimal, and some early compromises turned out to be bad decisions. There were a number times when it was clearly advantageous to throw away some earlier part of the implementation and re-code it; most of the time this didn’t happen because of the necessity of adding features and maintaining a running language system.

10.1.5 run-time code generation

As has been discussed in Section 6, Sol (like Scheme) is well suited to run-time code generation and modification. Unfortunately, the current implementation imposes restrictions on the use of run-time code generation, in that Sol applets cannot use eval without breaking the security restriction imposed by essentially all web-browser security managers. This is discussed further below.

10.1.6 portability

The Sol language system is based on open-source, freely available tools which have been ported to a variety of platforms. The Sol language system is known to work under Linux and the SGI IRIX operating systems, and should be easily portable elsewhere. Since Sol compiles to standard Java byte-code, compiled Sol applications and web applets should run almost everywhere.

In practice, Sol code is less portable than I had hoped. There are two major reasons for this. The first is code size; compiled Sol code requires the support of the Sol language system. This is not generally a problem for Sol applications, but it presents as serious problem for Sol applets which must be small in order to download quickly over slow links. A typical Sol applet, associated libraries, and core language system can easily exceed a megabyte of compiled code, making download times prohibitive.
The larger portability problem is that Java's promise of cross-platform compatibility is still largely unfulfilled. Significant differences in virtual machine implementations and language feature support make Java portability a “write once, test everywhere” proposition. This is particularly frustrating when working with the current Sol implementation, since there is often no obvious way of hacking around such compatibility problems without modifying the underlying Kawa compiler.

10.1.7 support

It's hard to describe a language as well supported if you are the only one using it. On the other hand, it's clear who to talk to if you want a new language feature implemented. Fortunately, the open-source tools on which the Sol language system is based are well supported through the open-source community. This type of support has been very valuable during the implementation process.

Ideally, the Sol language itself would become an open-source project. Such an outcome would tremendously improve the development process, and could turn the prototype implementation into a professional-quality development tool.

10.1.8 web applets and network programming

I have successfully used the current Sol implementation in creating web applets, including Growth, Orbit, RippleSort, and others. These applets are available for viewing through the Sol web-site, http://sol.media.mit.edu/SolDesign/. Problems with code size, restrictions on the use of eval, compilation quirks and portability issues have made me somewhat less than enthusiastic about building web applets with Sol.

I believe that the real strength of Sol as a networked procedural design programming language lies in the use of Sol to create networked applications and servers. Sol provide access to the functionality of the standard Java network packages, which should make this type of application relatively easy to code.
10.2 early interactive genetic programming experiments

My first experiment with genetic programming was a computational image breeding system I created as a Sol applet which ran locally under the Sun JDK Appletviewer. My original goal was to make this a web applet, but the security restrictions browser Java security manages place on creating new class-loaders prevented this from working.

The applet was functionally equivalent to the system described by Karl Sims in [24], but simpler. There were a number of problems in this implementation, but it proved the usefulness of Sol in an interactive genetic programming application. In particular, it demonstrated that sets and set operations work well to describe the genetic programming process. The image evolution applet and genetic programming library code went on to form the basis of my subsequent genetic programming experiments.

After creating the computational image system, I became interested in prospect of using genetic programming for evolving procedural image filters. Since a procedural image filter is simply a computational image which uses sampled data as a basis function, this was a relatively simple extension of my previous code. The result was the GPI system I described in Section 9.

10.3 GPI results

Creating GPI was a useful first step in exploring genetic programming, and demonstrated the strengths and weaknesses of interactive genetic programming as an artistic tool. One of the great benefits of genetic programming is that the process can be used by non-programmers; this point has been made many times. A less obvious, and perhaps more interesting result of my work with the GPI system was the discovery of how useful interactive genetic programming was for me, an experienced programmer with a knowledge of the technical side of digital image manipulation; interactive genetic programming allowed me to explore solutions I would never have thought to code myself.

The GPI system has many limitations, the most frustrating of which is the long rendering time (thirty seconds for a group of seven thumbnails on a fast machine is not uncommon), making the interactive cycle frustratingly slow. Slow rendering time also
places practical limits on population size, which limits the flexibility of the search process.

10.3.1 projects

My first project with GPI was to evolve procedure image filters for some low-resolution digital photographs of my hands. The input images had high contrast and a simple color scheme, as shown here. Even though the software was in an embryonic state, the filter images produced from the first runs were beautiful, if frustratingly slow to render. I have included some of the results here, and full color images may be found in Section 13.

My next project was an attempt to evolve a filter which would emphasize the background of a particular image. This was qualitatively different from the first project, because I had a goal more specific than producing an interesting or beautiful result. Because of the small population size (seven filters) it took several runs to produce a filter which I felt satisfied the requirements of my problem, but many other interesting filters were produced along the way.

The next major addition to GPI was the capability to work with multiple source images in a continuous image space, as described in Section 9. At this stage, I could only sample the image-space in perpendicular slices. The source images for this project were a stop-motion animated sequence. The filters I evolved for processing this sequence were interesting, but the results lacked the continuity needed for animation.

I then modified the GPI image-space sampling code to allow for arbitrary sampling of the image-space. Using these capabilities I was able to produce far more interesting filter effects than with the previous version, including the temporal warping effect I mention in Section 9.4.1. Through a combination of evolution and hand coding, I produced the filter which created the temporal warping image sequence in Section 13. In this project I recycled the stop-motion animation from the previous project, but produced a completely different result which was much more interesting as an animation.

With more filter evolution and tweaking, and a new set of input images I made two new animated sequences, excerpts from which...
I include here. It was in order to explain these images that I developed the image-space visualization applet which I describe in Section 9.

Finally I hard-coded a computational image selection function into GPI's genetic programming system. This was the first step in developing the APFA framework. The selection function improved the usefulness of the GPI system somewhat, but slow filter rendering times continued to prove very frustrating.

10.4 evolution++ results

The evolution++ system (described in Section 9 was created in response to the limitations of the GPI system. Breaking the application into a Sol genetic programming server and C++ rendering/GUI client dramatically improves filter rendering time, allowing the system to filter a live video feed at interactive rates.

Along with faster filter rendering speed comes the ability to display and interact with much larger populations; the evolution++ user interface allows the designer to work interactively with generation sizes on the order of hundreds of filters, rather than the limit of seven or so using GPI. The most important technical feature of the evolution++ system is the implementation of an enhanced genetic programming process using the APFA framework.

10.4.1 basic APFA framework

The hard-coded computational image selection function implemented in later versions of the GPI system lead to the development of the APFA function chain architecture described in Section 7. The selection functions were implemented entirely on the genetic programming server side and could only examine filter code, not the rendered results of applying the filter. The basic APFA framework works very well, but its flexibility is somewhat limited.

10.4.2 extended APFA framework

An examination of the theoretical limitations of the filter chain architecture lead me to the extended APFA framework described in Section 7.3. Like the basic APFA architecture it extends, the preliminary fitness function architecture is currently implemented
entirely on the genetic programming server side, and does not have access to rendered filtered images.

Although the extended APFA system is still a work in progress, the early results are encouraging. As discussed in Section 7.3, the framework provides greater flexibility than the function chain architecture and provides the user with more information and control over the preliminary fitness assessment. To date, no major projects have been done using the extended APFA framework, although a number of tests have been run using a live video feed.

10.4.3 projects

The evolution++ system is more recent than the GPI system, and has been primarily used in a real-time interactive video filtering context. The system can also be used for non-interactive video processing applications, as was done in the GoVaul project. This project is described below.

10.5 the GoVaul project — a collaboration

The GoVaul project, a 35 second processed animated sequence, is the result of using the evolution++ system in a collaborative effort with Golan Levin. Section 13. The focus of Golan’s work is creating digital interactive visual art and design which is both immediately understandable and infinitely masterable, the digital equivalent of the approachability and power of drawing; any child can pick up a pencil and immediately begin making marks with it, but a lifetime can be spent mastering the art of drawing. Golan’s work focuses both on the the means of interaction and the visual results, bringing together elements of HCI and graphic design.

The collaborative piece (dubbed “GoVaul” by our advisor Prof. Maeda) began as two sequences of animation totalling 1073 frames captured from two of Golan’s interactive art projects: floo and aurora. Golan created these sequences specifically for the project, documenting his own interaction with floo and aurora.

Using a sample from the first sequence, I spent several hours using the evolution++ system to evolving image filters to post-process Golan’s animation. I made several GP runs, each between seven and ten generations, evaluating thousands of filters. The re-
result was 94 filters I found particularly interesting. Golan and I then worked together to pick six to use for the final piece.

Once the filters were selected, I scripted a filter sequence to match the 35 seconds of animation, and then rendered the results in batch mode. The entire filtering process, from the time Golan provided the initial sample images to the finished rendering of 1073 frames, took less than eight hours.

The results of this project were particularly interesting, and a demonstration of the value of the emergent design process. I discuss these conclusions in more detail below, in Section 10.6.2.

10.5.1 enhanced genetic programming for GoVaul

The GoVaul project used a combination of two “top-level” preliminary fitness functions to guide the search. The first, no-comp-images, is actually a combination of four simpler fitness functions which allow the user to specify how many sampling operations (uses of hue, saturation, and value procedures) are necessary to classify a filter as a true image filter vs. a computational image (computational images are given a bad fitness score). The parameters passed to no-comp-images allow the user to specify separate target thresholds for each of the three filter expressions, as well as an overall count which must be met for the image filter to be approved. This type of control is important, because samples used to calculate the value expression may be more visually important than samples used to calculate hue or saturation.

The second, u-and-v, also a combination of four simpler fitness functions, allows the user to specify whether to require the presence of both $u$ and $v$ in each (or any) of the three expressions. Source code for both of these can be found in Section 9.8. The combination of no-comp-images and u-and-v has proven very effective in producing image filters that have a clear visual relationship to the source image, though not necessarily a simple one.

10.6 conclusions

The research described in this thesis has two components: a technological component based the new techniques of enhanced procedural design, and a design-theoretical component based on the emergent design philosophy.
10.6.1 technology

I have devoted most of this thesis to describing the technology. In part it is because the technological component is easier to describe and evaluate; its success or failure is a relatively well posed question. The more important reason is that without demonstrating the viability of the supporting technology there is no basis to support the emergent design process.

**Sol** I believe that Sol is a useful procedural design tool. To my knowledge, it is the only general purpose programming language created specifically with procedural design applications in mind, although a case could be made for Adobe PostScript. The Sol language provides most of the features I outline for an ideal procedural design language, but the poor performance of the current implementation is frustrating.

Any general purpose programming language may be used for procedural design, but a language like Sol or Scheme which supports run-time code generation and modification is necessary for genetic programming and enhanced genetic programming applications. Sol’s set and category notation makes this process particularly clean.

**GPI** The GPI system has been successful, both in providing a platform to explore interactive genetic programming and as a design tool. Working with the GPI system allowed me to develop the ideas for enhanced genetic programming which I later implemented in evolution++. GPI’s image-space sampling capabilities are interesting, and as far as I know unique in a computational image processing framework. GPI’s usefulness is limited by its slow performance and corresponding limits on generation size.

**evolution++** The evolution++ system is still a work in progress, but it represents many technical improvements over the GPI system. From a theoretical perspective, the most interesting feature is the enhanced genetic programming framework. The current implementation is still in the testing phase, but has proven to be a useful tool in managing the complexity of the interaction task.
Another very important feature of the system is its speed. The ability to filter a real-time video feed (320×400 pixels) at interactive rates (10+ fps) on desk-top hardware is an exciting feature, and the result is a system that can handle the interactive display of large filter populations in a manageable way.

The off-line, batch filtering capabilities of the evolution++ system are still simple, but useful. Although incomplete, the evolution++ system could form the foundation of a complete enhanced genetic programming tool for real-time and batch image processing.

10.6.2 the GoVaul project

The GoVaul project is particularly interesting in the context of this thesis, because it represents the first test of the emergent design process under real design production conditions with a specific, non-trivial design problem to solve. The project was done on short notice under tight time constraints, using an relatively early version of the evolution++ software.

Despite these hurdles, Prof. Maeda has said that he believes the results to be among the best work either Golan or I have produced. I believe there are several reasons for the success of this project:

1. The quality of the source imagery was high; Golan’s work provided a strong visual foundation.

2. The high performance architecture and fitness-agents enhancements of the evolution++ system allowed for the evaluation of thousands of filters, making for a very powerful search process.

3. The filters were evolved specifically to match the input images; the importance of this was demonstrated by a sequence for which I had no example frames, which did not turn out as well as the rest.

4. The dozen filters that were chosen were highly selected and deliberately synergistic; only filters which preserved the overall dynamic feel of Golan’s work and contributed genuine aesthetics were chosen for inclusion in the final piece. This
allowed the quality of both aspects of the work (Golan’s initial imagery and my filters) to show through.

To summarize, I believe that it was a combination of excellent content, powerful tools, and a clear understanding of the quality that each of us brought to the project which made it a success. The GoVaul project demonstrates that emergent design can work very well under the right circumstances, but by itself is not sufficient; the key ingredient in any successful design project is skilled designers.

10.6.3 design theory

In many ways, this project has been an exploration of the interaction between technology and design. As a design tool, the computer offers radically new possibilities, many of which remain largely unexplored.

My presentation of the emergent design process has been mostly by example and induction. Much more could be said about the process, its historical antecedents, and its justifications. I have avoided this for two reasons. First, it would make for a much larger document, and second, I believe that the emergent design process follows naturally from the use of enhanced procedural design tools. If these tools are available and designers use them, it will necessarily change the way they work and approach problems.

The non-procedural design process is alive and well, and will likely remain the dominant design methodology for many years to come. However, procedural design and enhanced procedural design are two distinctly digital possibilities, ways of working which could not exist without the computer.

Procedural design itself has only recently begun to be an accepted part of the visual design field; few designers have the requisite programming skills, and few programmers have a strong design background. Participatory procedural design tools can effectively open up the field of procedural design to a wider group of artists and designers. I believe that this thesis demonstrates that such tools are both feasible and useful, at least in one relatively narrow application; more work must be done to show that the emergent design methodology and tools outlined here have
broader applicability and that designers other than myself will use them.

10.7 implications

Late in the writing of this thesis, Prof. Maeda said, “Discuss your role as an artist vs. a technologist. What would you do if hundreds or thousands of people saw your tools and wanted to use them? Do you see yourself as the next Kai\textsuperscript{16}, producing emergent design tools for a mass market?” These are good questions, and strike at the heart of what it means to be both an artist and a researcher.

As an artist, I’m driven to create new tools for my own use. I developed the emergent design process, first and foremost, as an exploration of what enhanced interactive genetic programming could be used for in my own work. For my work as an artist, these tools are a means of using the medium of digital computing in a new way, in a way that takes me beyond the confines of what I have done before. What I have found is a vast territory waiting to be explored. Emergent design tools don’t make me a better designer, but they do allow me to explore a much wider range of possibilities far more quickly than would otherwise be possible; this could be a great advantage in the competitive world of design.

As a researcher, I have a great respect for the power of knowledge which derives from the free flow of ideas and information. My small contributions to the field of science and technology, whatever they may ultimately prove to be worth, are only possible because of the vast contributions of those who have gone before me. It is very important to me that I share my ideas and do my part to support the process which has so generously provided me with the tools of knowledge and understanding.

I believe the key to reconciling these two points of view is the understanding that the ideas and the process I have developed as a researcher is not the content I bring to the design process as an artist. If I were to keep my tools secret (by withholding this thesis, for instance) it might provide me a temporary advantage as a designer. Conceivably, keeping the emergent design process as a “trade secret” could even provide the decisive factor in my suc-

---

\textsuperscript{16}... of Kai’s Power Tools fame, popular set of image-effects plug-ins for Photoshop targeted at a the mass market.
cess vs. failure in the art world. However, this advantage would be short lived, for just as successful art spawns imitations successful technology spawns reverse-engineering. Inevitably someone else would figure out how to do what I do, or develop and even better process. Ultimately, my success as an artist it will rest on the strength of my design skills, not proprietary design technology.

I believe that my contribution lies both in the development of tools for my own benefit and the publishing of my ideas for the benefit of others. In the short run, I will still be the only one to have access to these tools; I have no intention of developing emergent design plug-ins for PhotoShop, and others are unlikely to go through the technically demanding process of implementing emergent design tools unless and until I demonstrate their success. In the long run, if these tools are successful, than others will develop them one way or another. In the mean time, I will have made my contribution to the research process and created some really cool images along the way.

10.8 future research

The work done for this thesis represents only the smallest part of what must be done in order to make emergent design or some other enhanced genetic programming design a well developed, widely used design methodology. To paraphrase Neil Gershenfeld, “interesting theses are never finished, they are abandoned.” Looking back at this body of work, there is a great deal more I would like to do.

The evolution++ system is still a work in progress. In particular the advanced APFA framework and associated GUI elements need further development. It is currently not possible to change the weighting of APFA functions through the GUI; adding this feature would greatly improve the flexibility of the tool. Likewise, thresholding should be added so that the designer can limit the view to the top n filters in a generation based on the current ranking.

The most interesting technical challenge, and the next major step is to investigate the use genetic programming techniques to evolve populations of APFA functions to match the selection criteria of a particular designer and design problem. This is a particularly exciting prospect, because if successful this technique could
produce enhanced genetic programming design tools that learn the designers goals and preferences without the necessity of human-coded APFA functions. Stated more simply, evolving APFA functions could obviate the need for human programming at any level in the design process, yet provide intelligent design tools capable of producing sophisticated procedural design solutions.
11 Bibliography

References


12 Appendix: The Sol Programming Language

In this appendix, I provide an overview of the Sol programming language and some of its important features. More information, including implementation details, can be found in [5].

12.1 Introduction

Sol is a dialect of Scheme LISP extended with concepts from set and category theory. Like SETL, Sol is inspired by the power and flexibility of sets, but goes further than SETL in the use of sets for expressing data-type and concurrency.

Sol was developed as a language for creating procedural design tools. My goal was to create a language that was as beautiful as the ideas I wanted to express, and which would provide a clean syntax for parallel operations. I also wanted to create a language that was practical, portable, and encouraged good software engineering practice.

Sol is important to this thesis because it is the language which forms the basis of much of my procedural design work while in the ACG, and is the language in which the evolution++ genetic programming system is implemented. This chapter describes the Sol language and some of the details of its implementation.

12.2 Sol Concepts

Just as lists are a natural representation for sequential operations, sets provide a natural syntax for expressing concurrent operations. E.g. evaluating a set of procedures in Sol results the parallel evaluation of each procedure. By combining sets and lists, Sol provides an elegant syntax for computation capable of describing a wide range of procedural forms.

---

17 Sol stands for Set Operations Language.
18 The term “set” has very strict mathematical requirements (it is difficult to prove that something is really a set) so formally speaking we should refer to Sol sets as classes. However, the term “class” has another conventional meaning when discussing programming languages. To avoid confusion, I will use the term set to refer to the mathematical entity class and class in the object-oriented programming sense of class.
12.2.1 Lists

Lists are a natural representation for sequential operation, and form the basis of the LISP (short for LISt Processing) programming languages. Scheme LISP was chosen as the starting point for Sol, because Scheme LISP provides a particularly elegant and syntax for expressing sequential computation. Sol supports all \( R^4 \) Scheme constructs\(^{19} \).

12.2.2 Sets

Like a Scheme list, a Sol set is a collection of things. Unlike a Scheme list, a Sol set can be either finite or infinite, ordered or unordered. Every Scheme list has an explicit external representation, but many Sol sets don’t have one, either because they are infinite (the set of complex numbers) or are the result of set operations which don’t reduce to an explicit set. Sol sets are hierarchical — just as COMPLEX is a subset of NUMBER, all Sol entities are also subsets of the SOL set.

Sol defines a special set of procedures called set operators. The process of applying a set operator (operator for short) to a set is called an operation, and every operation results in a new (possibly empty) set. Generally speaking, set operations do not alter the contents of existing sets.

Sol sets come in three basic flavors. There are the core sets, which form the basis of all other sets and are defined as part of the language environment. There are fundamental sets, a designation which includes the core sets, Java classes, and explicit user-defined sets. And there are derived sets, which are the result of set operations on fundamental or derived sets, e.g. (union boolean integer). Sets and set operations provide Sol with a very powerful and flexible typing system, as well as lexical parallelism — one can see concurrency in a Sol program simply by examining the source code.

Core Sets Core sets are so-called because they are part of the core Sol language specification, and represent the primitive Sol

\(^{19}\)Scheme is intended as the starting point, not a subset of the language. In the future Sol may be implemented as a stand-alone compiler or interpreter, in which case \( R^4 \) compatibility may be sacrificed for the sake of clarity and efficiency.
types, e.g. the set of integers, strings, lists, etc.. Every $R^4$ Scheme type has a corresponding Sol core set. In addition, there are a number of core sets relating to aspects of the Sol language not shared with Scheme (the set of sets, for instance). Core sets are special in that all other sets in some way derive from core sets, either by explicitly including members or through operations on other sets.

**Fundamental Sets**

This category includes core sets, Java classes, and user-defined explicit sets. These sets are called fundamental because they can be expressed without resorting to set operations on other sets, e.g. (set 'foo' 3.14159265 real (set 1 2 3)) which is explicit, <java.lang.String>, which is a Java class, or (union (set 1 2 3) (set 4 5 6)) which simplifies to (set 1 2 3 4 5 6).

**Derived Sets**

A derived set is a set resulting from one or more set operations on fundamental or derived sets which does not simplify to a fundamental set, e.g. (union (set 1 2 3) string) (the union of \{123\} and the set of strings) or (difference real integer) (the open set resulting from $\mathbb{R} - I$). Note that (union (difference real integer) integer) simplifies to real and hence is a fundamental, not a derived set.\(^2\)

**Data typing**

Data type in Sol is expressed through set membership. Any set (or set expression) can be used to specify type. Compound objects, such as lists or vectors, can be described in terms of the type of their elements. This can be seen in the syntax of the Sol function, which allows the name and type of each parameter to be specified, as well as the type of the return value. This mechanism of type specification is very powerful, since type constraints may be as tight or loose as the programmer wishes, e.g. (function integer ((a real) (b string) (c (set 1 "help!" 3))) ...) specifies an integer-valued function of a real, a string and a member of the set (set 1 "help!" 3), but (function

\(^2\)The current implementation of Sol does not perform this type of simplification automatically, so the set resulting from this statement may be considered a derived set.
sol ((a number) (b sol)) ... specifies a function of a number (may be real, complex, etc.) followed by any Sol object, returning any Sol object.

12.2.3 Categories

Sol categories provide many of the advantages of objects, but with Sol language orientation towards concurrency through set operations. Rather than starting from a formal description, let us consider the example of representing and manipulating a three-dimensional vector.

**Function vs. Object Oriented** In a functional language, a three-dimensional vector of reals (three-vector for short) could be a data structure holding three real numbers \((x y z)\) and a collection of functions operating on vectors \(((function1) (function2) \ldots)\). In an object-oriented programming environment one might define a three-vector class, combining the coordinates (the member data) with the vector functions (the member methods), *e.g.* \((x y z (method1) (method2) \ldots)\). The main advantages of the object-oriented approach are data abstraction, encapsulation, and inheritance. However, object orientation imposes certain limitations. For instance, when dealing with large numbers of instances, an array of records,

\[ ((x_1 y_1 z_1) (x_2 y_2 z_2) \ldots) \]

may be significantly more compact than the corresponding array of objects, especially if a function table is present:

\[ ((x_1 y_1 (method1) (method2) \ldots) (x_2 y_2 z_2 (method1) (method2) \ldots) \ldots) \]

More importantly, the data-procedure encapsulation present in object instances may make it slower to manipulate objects than a corresponding collection of records\(^{21}\). If many operations of this type are performed (such as translating or rotating every vector in an array) the overhead can be significant.

\(^{21}\)For instance, referencing member functions often requires two pointer dereferences: the first to get the location of the object and another to locate the function using the object’s function table.
Category as Container Class  To avoid this problem, one might design a new type of object (a container class) which holds an array of instances and methods for manipulating objects in the array. If all of the methods of interests are in the function table of the container class, than only one dereference is needed per call. Further, some of these methods (call them operators) may operate on the array as a unit, exploiting efficiencies of parallel operation unavailable to the member methods of the individual objects. Replace the array with a set, and this type of container class is essentially a Sol category.

Sol Categories  A Sol category contains a (possibly empty) set, zero or more member operators, zero or more member data, and zero or more member functions. A member operator is a procedure of at least two arguments (a set and a set element) which returns a new element of a new set. Category member operators can be applied in parallel to the member set, resulting in a new set.

Sol categories are similar to classes in a conventional object oriented programming language, providing data/method encapsulation and a single-inheritance mechanism. However, Sol categories differ from classes in a fundamental way; Rather than being a mechanism for encapsulating a heterogeneous collection of data and methods (a class), a category combines a single member set with methods for operating on that set in parallel, as well as more general methods which may operate on any data.

Through the inheritance mechanism “abstract” categories (categories containing empty sets) may be defined, which other “instance” categories inherit operators and methods from to acquire. Abstract categories can also serve as a useful organizational tool by providing named collections of static functions, just like static classes in a conventional object-oriented language.

12.3 The Sol Implementation

Sol is currently implemented as a combination of an embedded language in Scheme lisp and various Java classes, which function together through the the Kawa Scheme-to-Java bytecode compiler[3]. This implementation strategy was chosen to minimize development time and maximize portability. The primary drawback has
been in performance, which is why the computationally intensive portions of the \texttt{evolution++} system are written in C++.
selected images

early GPI images  These images were created using the first version of the GPI system during the first successful GP runs.
temporal warping  This image sequence was created using a later version of the GPI system, and illustrates the temporal warping effect.
GoVaul images I  Images from the GoVaul project.
GoVaul images II  Images from the GoVaul project.
GoVaul images III  Images from the GoVaul project.
GoVaul images IV  Images from the GoVaul project.