

Table of Contents

| | |
|--------------------------------------|----|
| Introduction..... | 3 |
| Project Goals..... | 4 |
| Genetic Programming..... | 4 |
| Background..... | 5 |
| Preliminaries..... | 6 |
| Genetic Programming System..... | 8 |
| Libraries..... | 8 |
| Processing Loop..... | 9 |
| ECJ System..... | 10 |
| Population Initialization..... | 10 |
| Crossover..... | 11 |
| Mutation..... | 11 |
| ADFs..... | 11 |
| Texture Rendering..... | 12 |
| Function and Terminal Sets..... | 12 |
| Return Types..... | 13 |
| Terminal Set..... | 13 |
| Function Set..... | 13 |
| Parameter Control..... | 15 |
| Fitness Evaluation..... | 15 |
| Response Values..... | 16 |
| Mean..... | 16 |
| Standard Deviation..... | 17 |
| Deviation from Normal..... | 17 |
| Direct Color Distance Matching..... | 17 |
| Direct Color Histogram Matching..... | 17 |
| Ranking..... | 17 |
| Features..... | 19 |
| GP Tree Viewing..... | 19 |
| Population Monitoring..... | 19 |
| Predefined Textures..... | 20 |
| Textured Primitives..... | 20 |
| Results..... | 21 |
| Conclusions..... | 27 |
| Future Work..... | 27 |
| Final Words..... | 27 |
| Appendix..... | 28 |
| Creation Process..... | 28 |
| Literature Search..... | 30 |
| User Manual..... | 35 |
| Getting Started..... | 35 |
| Main Window..... | 36 |
| Tree Display..... | 36 |
| Update Panel..... | 37 |
| Genetic Programming Parameters..... | 37 |
| Progress and Menu..... | 38 |
| Fitness Functions Window..... | 39 |
| Function/Terminal Set Window..... | 40 |
| Globals Window..... | 40 |
| General..... | 41 |
| Genetic Programming..... | 41 |
| Source Image Window..... | 41 |
| Progress Chart Window..... | 42 |
| Progress Stats..... | 42 |
| Population Stats..... | 42 |

Tree Viewer Window.....43

Introduction

This paper introduces JNetic Textures, an interactive genetic programming system with its interface framework based off of its predecessor, JNetic. This system was the final product of the COSC 5P75 reading course under the supervision of Professor Brian Ross. Although this paper focuses primarily on the system itself, it also outlines additional information that I've learned throughout the course, such as experiences with other systems, improvements, milestones and roadblocks.

JNetic Textures is a GP system which focuses on producing procedural textures based upon an extensive amount of user control. Because the final products are procedural textures, the system also focuses on the ability to re-size and save the textures into any format without any major image or detail loss. The system uses ECJ as a GP framework for evolving a population of textures, where each texture is represented by a typical GP tree. In order to generate the actual texture from the tree, each pixel in a result image must be given an RGB value which is attained by evaluating the GP tree (which returns an RGB value). Each node must be evaluated from the root down, returning a color value for a single pixel (x, y). This is done for all values of x and y from [0..width] and [0..height] in the image, respectively.

Each texture result in the population is scored based upon various evaluation functions, the set of which is chosen by the user. This is a multi-objective approach. Each function is automatically calculated so there is no real need for user intervention, though this of course is up to the user. Individuals are ranked individually using the rank-sum method, instead of the typical pareto ranking method, in order to avoid the problems with pareto fronts and large groups of individuals with the same rank.

This system differs from JNetic in a multitude of ways. Where JNetic used GA, this system uses GP. JNetic's image evaluation was based entirely on a per-pixel color distance calculation, whereas this system utilizes a multi-objective approach. Another notable difference is the removal of the chromosome editor. Color model editing was removed as well, which increased the search space size heavily due to the magnitude of the (256, 256, 256) color model. This was one of the most innovative portions of JNetic, and contributed greatly to the visual appeal of its images. Future versions of this system might incorporate something similar.

This system is based off of previous work done on the *Gentropy* system (Wiens & Ross 2002), using the framework of JNetic. The largest similarity to JNetic, however, is the subjectivity of the value of the final images. Since the topic of art is such a subjective area, it is difficult for a system to assign an image with a 'good' or 'bad' tag without any debate from outside parties. Thus, it is entirely up to the user as to what they feel is a good final product; sometimes the most interesting textures are the ones randomly generated in generation zero.

Steve Bergen

Project Goals

The primary goals of this project – or reading course – were as follows :

1. *Gain experience using a genetic programming system, and all the functionality behind some of the more advanced features (ADFs, multi-objective)*
2. *Research existing material focusing on genetic programming, and its applications to the topic of texture generation*
3. *Produce a texture generation system similar to Gentropy, using the framework of JNetic and the existing fitness evaluation functions in Gentropy*
4. *Allocate large amounts of control to the user in the new system, allowing easy access to parameter, individual and population data*

In addition, other more secondary goals were :

1. *Implement textured geometric shapes into texture generation*
2. *Gain experience with multi-objective fitness evaluation*
3. *Generate population data with respect to fitnesses in a graphical format*

I also took time to look into research papers detailing more texture methods in order to add to the existing function set. Material was also read that pertained to genetic programming and structural design, for the use of my thesis. These papers in particular were read early in the course in order to contribute to genetic programming research, instead of focusing on structural design.

These main goals were fulfilled, and in addition other goals came up throughout the course. Some of these were implemented, such as JNetic's color distance evaluation, while others were attempted but proved to be either too time-consuming, difficult or produced sub-optimal results.

Genetic Programming

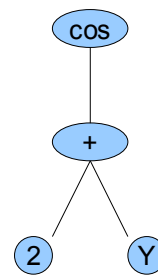
Genetic programming is an evolutionary computation technique first created by Koza, in order to evolve a population of computer programs (Koza 1992). In this way, it differs greatly from genetic algorithms, but other than this fact they are quite similar.

Genetic programming (GP) gets its basis from the foundations of evolution. The general concept is that individuals, over time, have the capacity to evolve to better suit their environment and other contributing factors. This same goal exists with GP, as we attempt to evolve computer programs with the hopes that as time goes by, their worth, or fitness, will improve. With natural evolution, this is achieved with the concept of survival of the fittest, or natural selection. This concept generally means that the more fit individual will likely have a larger chance to reproduce, and therefore pass on its dominant features to its offspring. As this process continues, the population will get more and more fit, and thus adapt to their environment (Poli & Langdon 2009).

This process of reproduction is a key factor in evolutionary computation, or more specifically with genetic algorithms and genetic programming. Before reproduction can take place, parents must be selected. Most often, the most fit parents are selected, but this is not always the case. In order to introduce new and potentially beneficial genes or traits into the population, occasionally less fit individuals need to reproduce as well. Once two parents are chosen, crossover occurs, splitting up the parent chromosomes and combining them to form child chromosomes. In this way, the traits of the parents are passed on to the offspring. In

addition, mutation also occurs in order to introduce new (and sometimes unwanted) traits into the population. The process of selection, crossover and mutation is known as reproduction.

In genetic programming, in order to determine the fitness of an individual, its program needs to be executed. These programs are stored in tree data structures, where nodes that have children are known as non-terminals, whereas leaf nodes are terminals. Normally, non-terminals are functions which process their children nodes as parameters, and terminals are numerical values of variables.



GP Tree example

$\cos (2 + y)$

Evaluated as :

Cos
+
2
y

For example, the function $\sin(x)$ takes a single value as a parameter, and so this value would be the child of the non-terminal node \sin . When generating a GP tree, all node constraints must be satisfied. This means that every node has a return type, and a node cannot be assigned as a child of another node if the child type differs from the return type. For example, if $\text{add}(x, y)$ returns an integer, and takes in two integers as x and y values, then the children of x must return integers. Note that we can also string together non-terminals as children of non-terminals, such as in $\sin(\cos(\tan(x)))$.

Once the tree's program is executed, it passes through a fitness function. This function is normally created specifically for a particular problem, and so will differ for each problem. Various crossover and mutation techniques exist for GP. The general GP process is as follows :

Initialize population

Evaluate population

While the current_generation < max_generation do :

While new_population_size < population_size do :

Select 2 parents via selection mechanism

Perform crossover with probability P_c

Perform mutation with probability M_c

Replace population with new_population

Evaluate population

Increase current_generation + 1

Most implementations of genetic programming systems contained within an interface have an extra 'update' step after the start of the first loop and before the start of the next. This step allows the programmer to update the interface with the current population, save data or any other needed operations. This step has no affect on the algorithm itself, but is used more for user interaction and monitoring purposes.

Background

This system was inspired and based heavily off of previous work done on *Gentropy* by Ross and Wiens. Gentropy produced 2D procedural textures using a similar function and terminal set with automatic fitness calculation, but lacked the user interface this system has. Thus, at the end of a run, the outputs were image and data files, and there was no form of user interaction or monitoring. This is one of the major improvements that was set as a goal in this project's early stages. Gentropy used lilgp as its GP system,

JNetic Textures

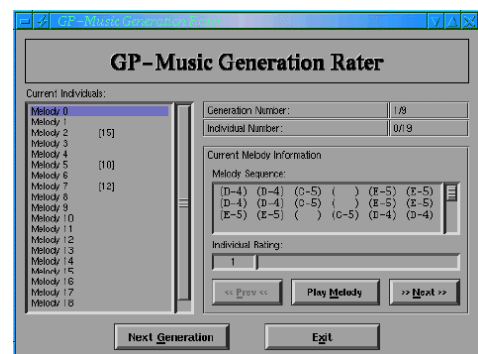
which was C-based. JNetic Textures used ECJ, which is based heavily off of lilgp but Java-based. Another system by Ross was used to evolve artistic filters from existing images using genetic programming (Neufeld, Ross & Ralph 2007). IMAGENE was another system, similar to Gentropy, that evolved procedural textures using genetic programming, though their function and terminal sets were small (Xu, D'Souza & Ciesielski 2007).

Two-dimensional procedural textures have been used in the past for a variety of purposes, but find their greatest use in 3D modeling and rendering. Their application to 3D modeling is highly valued over other conventional bitmap textures due to their ability to be re-sized, stretched and reformed with little to no loss of detail, and the wide variety of existing texture rendering functions available (Ebert, Musgrave, Peachey, Perlin & Worley 2002). Due to the mathematical nature behind them, they are relatively easy to compute and require little storage space as well. Three-dimensional procedural textures have been recently applied to models as well, forming bumps, ridges or even more complex patterns (Bhat, Ingram & Turk 2004). Kopf used a technique of combining multiple 2D procedural textures with a layering technique in order to produce more crisp textures on a 3D model, as well as produce 3D textures from 2D images (Kopf, Fu, Cohen-Or, Deussen, Lischinski & Wong 2007). This process produced great results.

Brooks produced a system that produced textures based upon user interaction (Brooks & Dodgson 2005). This system allowed a user to first select an existing texture, then make modifications to it via color-changing and transformation operators. The system then attempted to replicate the changes across the entire texture. The idea was to have the user edit only a small portion of a texture, but carry out that change across the whole image. This also allowed users to texture 2D images. The results produced were interesting, but not too novel.

JNetic was a system created to produce vectorized versions of source images using genetic algorithms (Bergen 2009). The system was reinforced by a strong user interface element, allowing direct chromosome editing and extensive parameter controls and population management tools. The results produced were great, and the interface framework is reused in this project.

GP Music was an attempt to attach a user interface to a GP system, though focused more on replacing the subjective fitness evaluation of the user with an automatic one (Johanson & Poli 1998). This system evolved a population of music samples composed of notes, and required the user select fitnesses for each individual in early generations. In later generations, a neural network was used to analyze the user's previous choice and take over fitness calculation for the remainder of the run. The interface of this system was limited to simply fitness evaluation mechanisms.



Preliminaries

Procedural Textures

A procedural texture is a texture that is generated by some mathematical formula, rendered on a per-pixel basis in an image. Generally, the rendering of one pixel should not be influenced by the previous one, and because of this a true procedural texture will always look the same no matter what order the pixels are rendered in. Because of the mathematical nature of the functions used to generate the image, a procedural texture can also be re-sized to any scale, with little to no damage to the final product.

JNetic Textures

This is achieved using the concept of a (u, v) coordinate system. These values differ from an (x, y) pixel coordinate system in one way. Usually, u and v values are floating point values, where x and y pixel coordinates are always integers. The reason for the use of floating point for u and v is because most mathematical functions respond to more precise values, such as the cosine and sine functions. Therefore, when we create a procedural texture function, we use u and v as variables instead of x and y.

Let's look at a simple function, using u and v in place of x and y. A single pixel needs three values; red, green and blue. Each is in the range of [0.0..1.0]. Let's say we want to render our texture onto a 100 x 120 pixel image. So x and y range from [0..width-1] and [0..height-1].

Now we need to choose our u and v ranges. These could be really any numbers, but typically u and v range from [0.0..1.0]. Next we choose a step size for u and v. This value is very important, because it must scale with the image size. For our 100 x 120 pixel image, our u step size is

$$(1.0 - 0.0) / 100$$

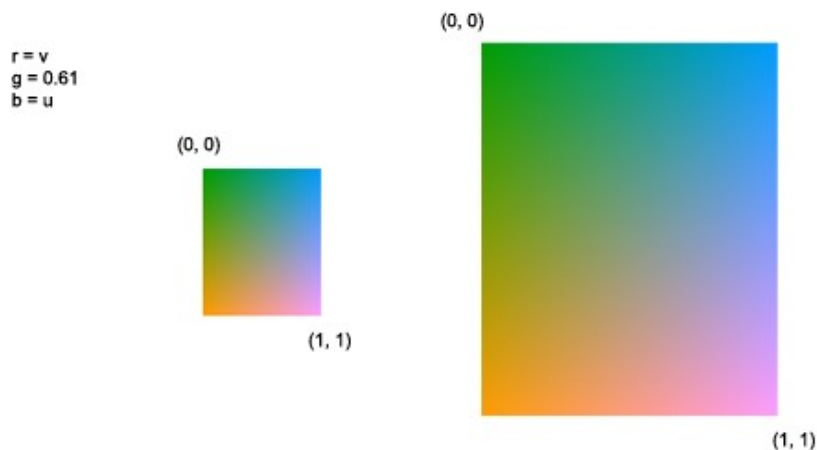
and our v step size is

$$(1.0 - 0.0) / 120$$

Now, if we want to render the texture onto an image, we calculate the RGB value for every pixel (x, y) using the corresponding (u, v) value.

$$u = x * u_step_size \quad v = y * v_step_size$$

If we want to increase or decrease the image size, the only thing that needs to change is the step size. This is the way textures are rendered in JNetic Textures, both for geometric primitives and backgrounds.



Genetic Programming System

This section describes in detail the implementation of the GP system on a java platform by extending JNetic's existing framework and using ECJ as the evolutionary system. The libraries section describes the java packages used during the course of this project, and the ones that are utilized in the actual system.

Libraries

A number of libraries were used in this system. Below are the ones I've experimented with and the ones I've actually added to the system.

ECJ (Luke)

ECJ is a java-based evolutionary computational system that focuses mainly on genetic programming, although genetic algorithms are also included. This system has a very steep learning curve, but in the end is extremely flexible. Some changes had to be made in order for the system to be of any use for this problem, however.

Java (Sun Microsystems)

This was more a tactical choice than anything. It is becoming more widespread knowledge that Java is quickly gaining on C and C++ with respect to speed and use. Since my past experiences thrived with the use of Java, I continue – and likely will always continue – to use Java when applicable.

JFreeChart

A java-based graphing system, which is used here for graphing population statistics.

JGAP (Gibson)

Much time was spent using JGAP. Advertised as a more easy-to-use system good for most problems, this was a first choice for GP systems. Unfortunately, it has its flaws. With no mutation operators and questionable crossover operations, this system was scrapped in exchange for ECJ.

JNetic (Bergen)

The decision to use my own system was an easy one, since it handles most of the interface programming already. A lot of the sub-windows needed to be scrapped to make way for genetic programming parameters, much to my dismay. One particular major loss was the chromosome editor. JNetic also contains several personally-created image utility tools which came in handy here.

JTexGen (RotStan)

This free java library is used to produce a variety of procedural textures, most of which are based on the usage of Perlin noise and wave theory. The idea to use a package like this came from attempting to implement a Mandelbrot set as a texture. Realizing that only a small fraction of the textures looked good, I thought it better to use a package that found some of the best-looking textures already, and allowed the user

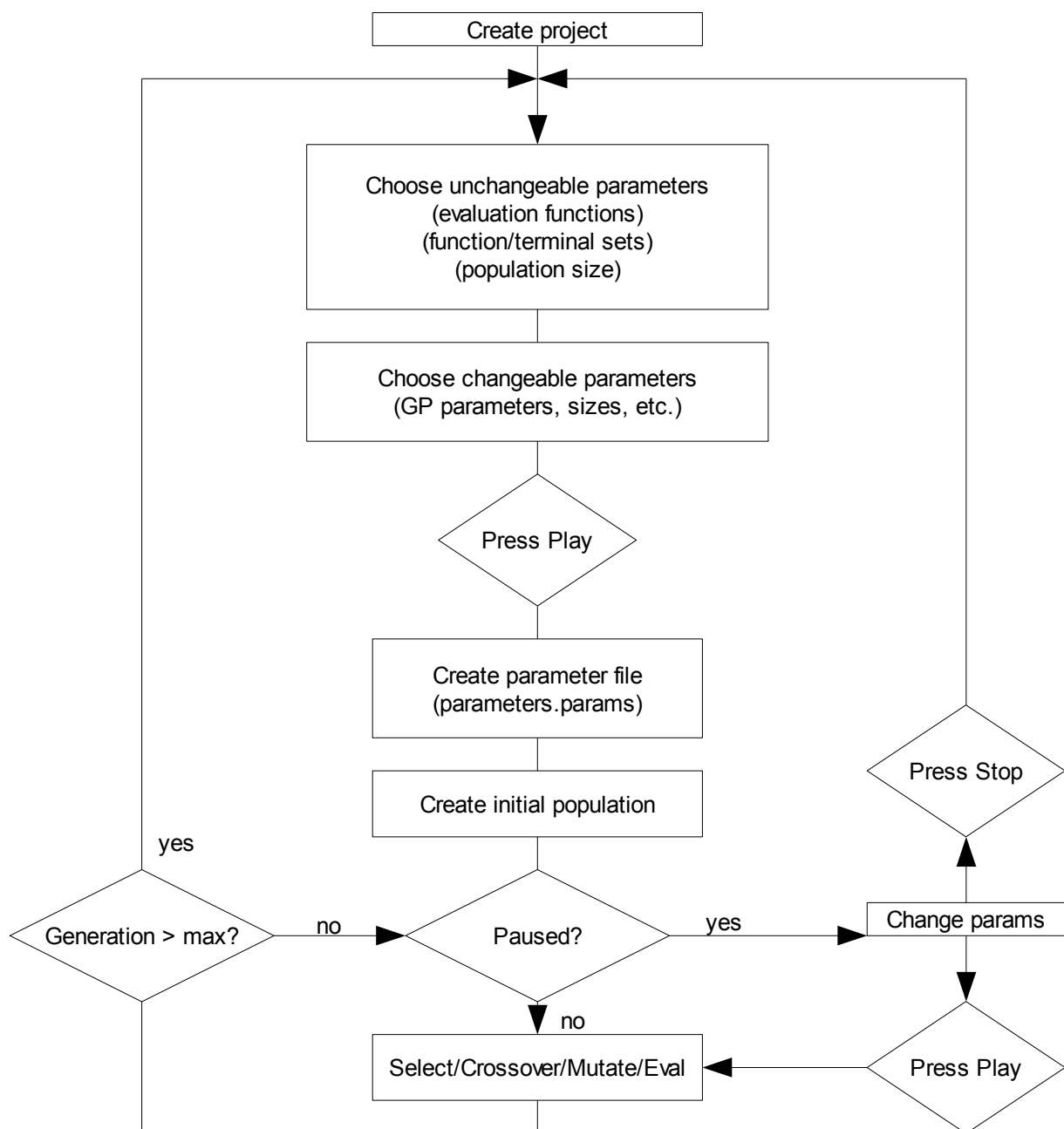
to tweak them with simple parameters. This library is used to generate the textures such as fire and grass.

Swing (Sun Microsystems)

Java's Swing library came in handy previously in JNetic's creation, and is used again here.

Processing Loop

The following diagram illustrates the generalized processing loop, from project creation to the eventual terminal of ECJ's generational loop:



JNetic Textures

The JNetic Texture system utilizes the general framework of JNetic, but there are large differences in the deeper processing loop. JNetic's genetic algorithm system was custom-built, and so control could easily be passed around. With the addition of ECJ's genetic programming system, the passing of control to and from the user was made a little more difficult, and thus in order to create a similar processing loop many changes had to be made to ECJ internally. Luckily, this had no affect on the performance of the actual system, and the end result was similar to JNetic with respect to the interface and user controls.

The user first selects a project name and a source image. The source image is for use in fitness evaluation, and the user must be careful about the image that is used (more on this in the *Fitness evaluation* section). Once this is complete, the user can choose from a number of parameters (more on this in the *Parameters* section). Some parameters cannot be changed once the GP has begun.

Once the user presses play, the GP can begin. At any time, the user can pause the loop. The pause does not take effect until the current fitness evaluation phase has completed. Once paused, the user can change certain parameters, resume the GP, or restart by pressing stop. Once stop has been pressed, the GP starts at generation 0 and the user has access to all parameters once more.

In JNetic, the best images were saved periodically to the project folder. This feature was removed in this system, due to the fact that it is difficult to differentiate between one texture and another with respect to fitness as evaluation is multi-objective .

ECJ System

The ECJ system contains a very powerful GP tool. It finds its strengths in its pipe-lined system, allowing users who want to alter aspects of the processing loop to do so. Unfortunately, this process is highly difficult as the system has a very steep learning curve.

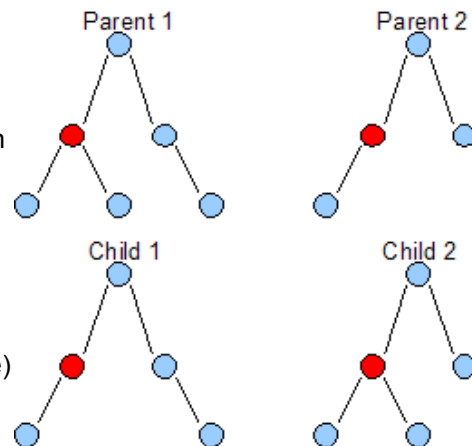
Some of the system's genetic operators were left unchanged, but a few were altered because they were either unreliable or insufficient for this problem. The genetic programming loop of ECJ is exactly the same as described in the *Genetic Programming* section. Each step in the loop, and its significance in ECJ is described here:

Population Initialization

The population is created with *ramped half-and-half*. The user can control all parameters for these, such as maximum and minimum depth, as well as the grow probability. Often, due to a weak pressure on the choosing of terminals and non-terminals, the tree produced will exceed or not meet the tree depth requirements. This is an internal ECJ issue, which on occasion will cause Java heap space errors.

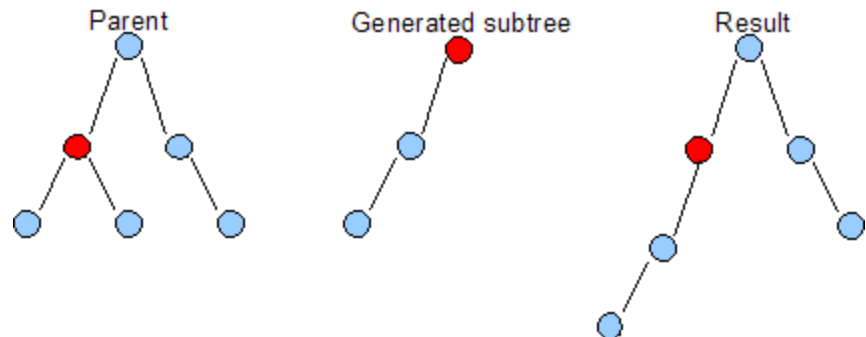
Crossover

One-point sub-tree crossover is used here. This had to be slightly modified from the original algorithm. The original algorithm seemed to favor certain nodes, instead of choosing all probabilistically. The algorithm also did not ensure that both chosen nodes from each parent were of the same return type. In this case, the algorithm decided that crossover will not occur, and replication will suffice. This led to very little population diversity after the first generation, and so the algorithm was altered to ensure that both nodes were of the same type, and that crossover would always occur (if possible) between two parent trees.



Mutation

Mutation initially wasn't done in ECJ, and so it had to be added. Sub-tree mutation was implemented for this project, which chooses a node in a child based upon some probability, then generates a new sub-tree at that node. In this system, mutation uses the grow method to create its trees; the same one used in initialization.

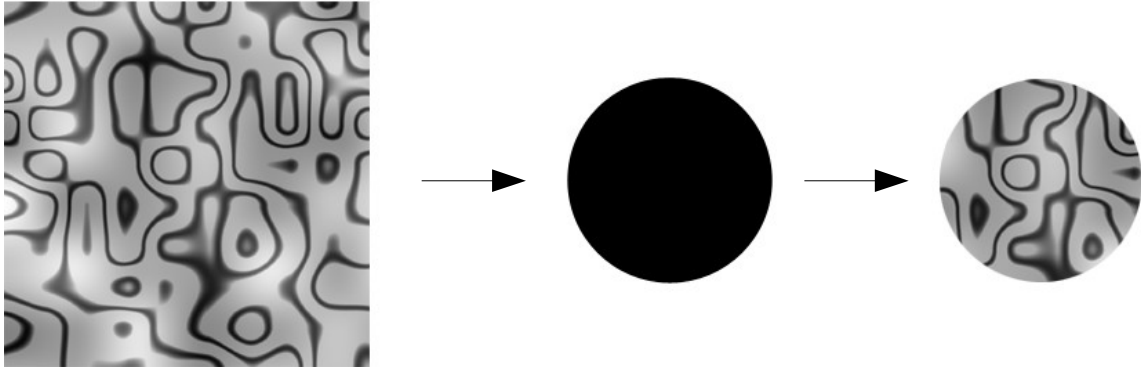


ADFs

Another great feature of ECJ is its use of ADFs. Any number of ADFs can be defined for a problem, where each individual's ADF can be placed over any node in the same individual's tree with the same return type. In ECJ, ADFs can also have their own parameters, though this feature is not used for this project. In JNetic Textures, an ADF returns an RGB value, and so can use any mathematical functions in the function set, as well as textures. Note that ADFs cannot create geometric primitives.

Texture Rendering

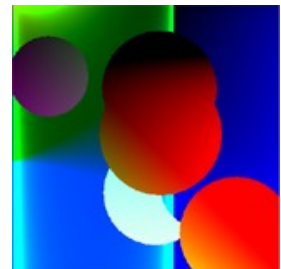
As explained previously in the *Preliminaries* section, textures are rendered pixel-by-pixel for an entire image by parsing the tree and evaluating every node. With the addition of geometric shapes, we need to evaluate each shape's texture separately.



For each shape, we need to

1. Create the texture over the entire image, but save it instead of overlapping it by rendering
2. Create the shape, once again saving in and not rendering
3. Apply the texture to the shape
4. Save the shape

By doing this, we can achieve very interesting results, especially if two shapes have the same textures. In the case of the image to the right, we have two different circles with the same background texture, which is in the form of an ADF. By stretching the texture over the breadth of the whole image, both shapes can 'share' the texture and appear to be combined instead of overlapped.



Function and Terminal Sets

In order to facilitate extreme diversity amongst individuals, a large and complex function and terminal set was implemented. Every function and terminal in the set has a return type and a number of parameters, or *children*. In order for a function or terminal to be compatible with one another, the return type of the parameter must be the same as the parameter type. This is typical in genetic programming.

Due to the wide range of the sets and the return types, I had initially hoped to reduce the size of the search space. For example, by introducing integer ephemeral random constants as different return types in the tree – despite the fact that they are all indeed integers – I can limit which functions can use which constants. This is the step I took when creating the terminals such as *radius* and *width*. In ECJ, we use a Data Object in order to pass information to and from nodes in the tree, and this is where the actual type of each terminal and function comes into play (integers, doubles, geometric primitives, colors). The return type names we use below are just a formality to satisfy node constraints.

Return Types

| | |
|---------|--|
| render | - The tree's root type. Every tree must have a root node of type render. |
| int | - A standard integer type. Integers are used exclusively with shape parameters. |
| double | - A standard double type. Doubles are used exclusively when generating RGB components. |
| rgb | - A red/green/blue 3-tuple. |
| shape | - A geometric primitive object. |
| radius | - An integer with the range of [0..R] |
| x | - This is an integer with the range of [0..image width – 1] |
| y | - This is an integer with the range of [0..image height – 1] |
| texture | - A pre-defined texture generated by JTexGen. |
| doublep | - A 'parameter double' that has no functions with the same return type. This is used in <i>texture</i> types in order to ensure uniformity as a texture is rendered. |
| ldouble | - A large double value, with the same purpose as <i>doublep</i> . |
| lint | - A large integer value, with the same purpose as <i>doublep</i> . |

Terminal Set

| | |
|---------|---------------------------|
| Int | (return int) |
| Double | (return double) |
| Radius | (return radius) |
| X | (return x) |
| Y | (return y) |
| U | (return double) |
| V | (return double) |
| DoubleP | (return doublep) |
| LDouble | (return ldouble) |
| LInt | (return lint) |
| Grass | (return texture) |

Function Set

| Name | Returns | Parameters | Explanation |
|----------------|---------|------------|---|
| render | render | rgb, shape | Renders a background texture, then a shape object |
| render-texture | render | rgb | Renders only the background texture (no shapes) |

JNetic Textures

| | | | |
|-----------------|---------|---|--|
| rgb | rgb | double, double, double | Returns an RGB 3-tuple |
| texturetorgb | rgb | texture | Convert a texture object to an rgb object (at current u/v coordinates). |
| circle | shape | x, y, radius, doublep, rgb | Draws a circle with specified parameters and background texture. Doublep specifies the alpha value. |
| circlelink | shape | x, y, radius, doublep, rgb, shape | Draws a circle with specified parameters and background texture. Doublep specifies the alpha value. Also draws an additional shape. |
| rectangle | shape | x, y, radius, radius, doublep, rgb | Draws a rectangle with specified parameters and background texture. Doublep specifies the alpha value. |
| rectanglelink | shape | x, y, radius, radius, doublep, rgb, shape | Draws a rectangle with specified parameters and background texture. Doublep specifies the alpha value. Also draws an additional shape. |
| triangle | shape | x, y, x, y, x, y, radius, doublep, rgb | Draws a triangle with specified parameters and background texture. Doublep specifies the alpha value. |
| trianglelink | shape | x, y, x, y, x, y, radius, doublep, rgb, shape | Draws a triangle with specified parameters and background texture. Doublep specifies the alpha value. Also draws an additional shape. |
| add | double | double, double | Mathematical add |
| sub | double | double, double | Mathematical subtract |
| mul | double | double, double | Mathematical multiply |
| div | double | double, double | Mathematical divide |
| pow | double | double, double | Mathematical power function (x^n) |
| mod | double | double, double | Mathematical modulus function ($x \% n$) |
| sin | double | double | Mathematical sine |
| cos | double | double | Mathematical cosine |
| log | double | double | Mathematical logarithm |
| noise | double | double | Perlin noise function (parameter is length) (Perlin 2002) |
| texturemix | texture | texture, texture | Mixes two textures together |
| texturecolormix | texture | texture, rgb | Mixes a texture and a color together |
| fur | texture | texture | Creates fur from an existing texture |
| mandelbrot | texture | doublep, doublep, doublep | Draws a mandelbrot with 2 parameters, and the third is alpha |
| noise | texture | ldouble, doublep | Draws noise with the noise function above, but |

JNetic Textures

| | | | |
|--------|---------|------------------------------------|--|
| marble | texture | lint, lint, lint, lint, doublep | slightly different. The third parameter is alpha. Draws a marble effect using wavelength parameters. The third parameter is alpha. |
| fire | texture | lint, lint, lint, lint, doublep | Draws a fire effect using wavelength parameters. The third parameter is alpha. |

Automatically defined functions (ADFs) only use rgb components for their root. Because of this, they cannot generate shapes, but only background textures.

Parameter Control

There are a wide variety of parameters to control in this system. These parameters fall into three categories :

1. System parameters
2. Pre-run parameters
3. Run-time parameters

The system parameters define how the system runs, and any tweaks the user may want to make to the background. These parameters are usually found in the *Globals* window, and include image sizes, threading and so on. There are not too many system parameters, but these can be edited at any time.

The run-time parameters define parameters that need to be initialized prior to pressing the play button. These are found in the *Fitness Functions* and *Function and Terminal Sets* windows. A number of these also exist in the *Globals* window, such as anything under the *genetic programming* tab.

During-run parameters can be edited any time, and usually include any system parameters, and those controls found in the main window.

Each of these parameters is stored in java storage classes inside the 'param' package. Once the play button has been pressed, the system creates a file called *parameters.params*, and stores this file in the project folder of the current project. This file contains all parameters that are used by the system – contained in the storage classes – in a pre-defined format. This file is only created once during a single run – once play has been pressed – and so any changes made during or after a run has no effect. Similarly, this file is only read into the system once – when play has been pressed. Because of the way the system is threaded, it is difficult to locate specific used variables during run-time, and that is why a parameter file is used. This is also the reason for some of the parameters being set to pre-run instead of run-time. In later versions, this might be fixed.

Descriptions of each of the separate parameters can be found in the help file.

Fitness Evaluation

Fitness evaluation takes place once after the initial population is created, and once for each generation after reproduction takes place. Fitness calculation is purely automatic, requiring no user input in order to rank individuals. JNetic Textures incorporates a multi-objective fitness evaluation, and so the user is encouraged to choose between 2-4 fitness functions. Each fitness function is carried out upon an individual during evaluation, and so the more fitness functions chosen, the longer evaluation will be. If, however, the user

wishes to only use one fitness function the system will still work.

Each fitness function is expressed in two ways; by a raw fitness value and an error fitness value.

Raw fitness value : This is the real value returned by the fitness function. For example, if we are using *mean* as our fitness function, the raw fitness value would be the mean of the image.

Error fitness value : Some fitness functions rely on a comparison value in order to calculate fitness. These functions are *mean*, *standard deviation* and *deviation from normal*. The error fitness is the absolute difference between the raw fitness value and the comparison value. For example, if an individual's raw mean value is 2.3 and our ideal individual should have a mean of 3.0, then its error fitness is 0.7.

Every individual stores both these values, but for the last two fitness functions they will be exactly the same.

Response Values

In addition, the first three fitness functions rely on response values, which are calculated for each new image. Image responses are used to calculate the Deviation from Normality, which is an automatic image evaluation function (Ross, Ralph, Zong 2006). The response of an image is calculated in order to generate the image's bell curve gradient, which is used to compare changes in color to other gradients. The following is calculated for each pixel (i, j) red value of an image:

$$|\nabla r_{i,j}|^2 = \frac{(r_{i,j} - r_{i+1,j+1})^2 + (r_{i+1,j} - r_{i,j+1})^2}{d^2}$$

Similar calculations are done for green and blue. Then, the stimulus of the pixel is calculated as:

$$S_{i,j} = \sqrt{|\nabla r_{i,j}|^2 + |\nabla g_{i,j}|^2 + |\nabla b_{i,j}|^2}$$

The response of each pixel is computed as :

$$R_{i,j} = \log(S_{i,j}/S_0)$$

where $S_0 = 2$. Internal measures are taken to ensure that values too large or small are ignored. The response values of the source image are saved at the creation of a new project, and are calculated for each image as needed.

The five actual fitness functions implemented are :

Mean

This is the mean response value of an image, and is calculated as :

$$u = \frac{\sum (R_{i,j})^2}{\sum R_{i,j}}$$

This value is compared to a comparison value chosen by the user, and thus the error and raw fitness values will differ. This is an error-minimizing function.

Standard Deviation

The standard deviation is calculated as :

$$\sigma^2 = \frac{\sum R_{i,j} (R_{i,j} - \mu)^2}{\sum R_{i,j}}$$

This value is compared to a comparison value chosen by the user, and thus the error and raw fitness values will differ. This is an error-minimizing function.

Deviation from Normal

The DFN value is a measure of the deviation from the actual and expected distribution of response values. The closer a distribution is to a normal distribution (calculated from the mean and standard deviation values), the better the score. This value is calculated as :

$$DFN = 1000 \sum p_i \log\left(\frac{p_i}{q_i}\right)$$

The value of p_i is the observed probability in the i th bin of the histogram, and q_i is the expected probability. Low DFN values are preferred in this case. This value is compared to a comparison value chosen by the user, and thus the error and raw fitness values will differ. This is an error-minimizing function.

Direct Color Distance Matching

Direct color distance matching is taken directly from JNetic and computes a score from [0.0..1.0] reflecting how close one image is to another. A score of 1.0 is a perfect match, where 0.0 is an exact opposite. To do this, the color distance between two pixels is calculated using the 3D distance formula using red, green and blue color values. This is done for every pixel in the image, then the values are summed and divided by the total pixels in the image. This fitness function's raw and error fitnesses are the same. This is an error-minimizing function.

Direct Color Histogram Matching

Direct color histogram matching involves calculating the distance between the color histograms of two images. In order to create an image's color histogram, each pixel's color is quantized, resulting in a color value from 0 to 511. The number of occurrences in the image of each value is summed up and stored in a histogram. To compare two histograms, the absolute difference for each bin in the histogram is calculated and summed. A distance of 0 is a perfect match. This fitness function's raw and error fitnesses are the same. This is an error-maximizing function (the error corresponds to color distance where a distance of 1.0 is a perfect match, so an error of 1.0 is perfect).

Ranking

Each texture in the population is ranked using the *rank-sum* method. Initially, *pareto ranking* was used to rank the population, but this led to a small number of large pareto non-dominated fronts, which meant every individual in a front had the same fitness, or rank.

In the rank-sum method, every individual's fitness value is ranked with respect to the rest of the population. Once all the fitness functions have been ranked, these ranks are summed up and this new rank corresponds to the individual's new single fitness value. The basic algorithm is below :

```

for each fitness function  $f$  :
    sort population by  $f$  (best to worst)
    rank = 0
    for each individual  $I$  in population :
        rank_ $f$  = rank
        if the last individual's fitness is not the same, then rank = rank + 1
        ... otherwise rank does not increase
    for each individual  $I$  in population :
        fitness = 0
        for each fitness function  $f$  :
            fitness = fitness + rank_ $f$ 

```

Using this method, the best individual will have the lowest fitness rank. This method also avoids grouping individuals together despite large differences in fitness. For example, if individual A has fitnesses 0 and 999, fitness B has fitnesses 1 and 1, and individual C has fitnesses 0 and 2000, since neither one dominates the other they will end up in with the same rank with pareto ranking. With rank-sum, however, individual A will have rank 1, B will have rank 1, and C will have rank 2.

Features

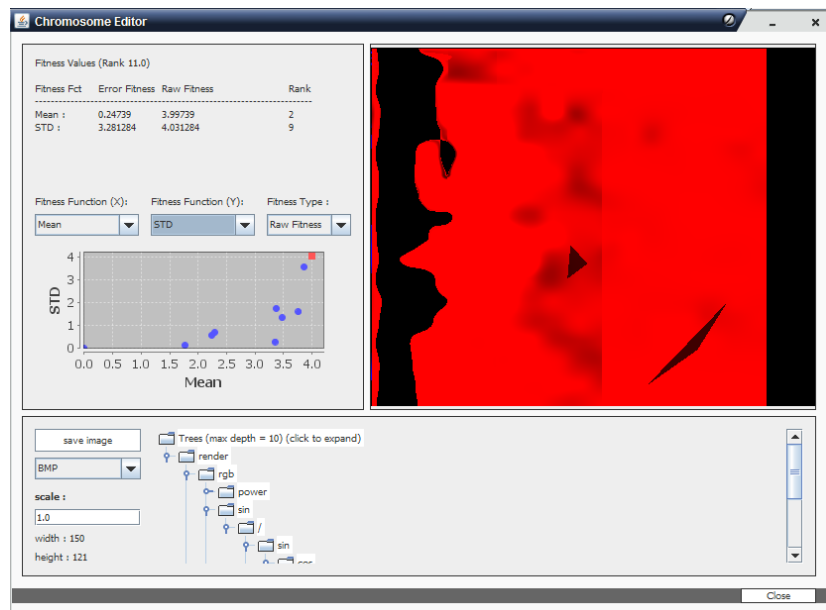
GP Tree Viewing

The GP Tree viewer was created to replace the chromosome editor in JNetic. The purpose of the viewer is to allow the user to get more detailed information about the tree without cluttering the main window.

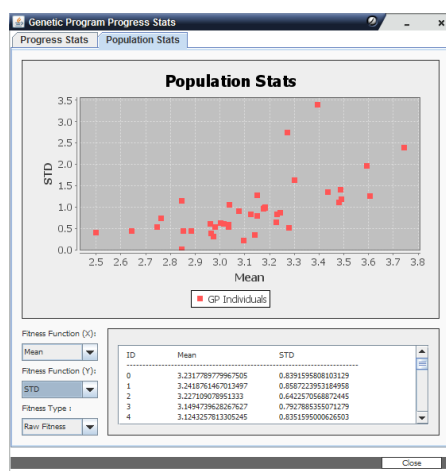
Displayed here is the texture at full size, alongside two other panels. The panel on the left displays information related to fitness, such as the actual and error fitnesses for each fitness function used. It also displays the ranks with respect to the population of each function. Below this information is a graph that shows this individual's placement with respect to the fitness

of the rest of the population. The user can choose the X and Y axis fitness functions, as well as choose the type of data that is graphed (raw or error).

The bottom panel displays the tree in its entirety as a Java list-tree. Folders represent functions, and when clicked show all the parameters of that function. This also displays any ADFs. The user can also save this image by scaling it to new dimensions.



Population Monitoring

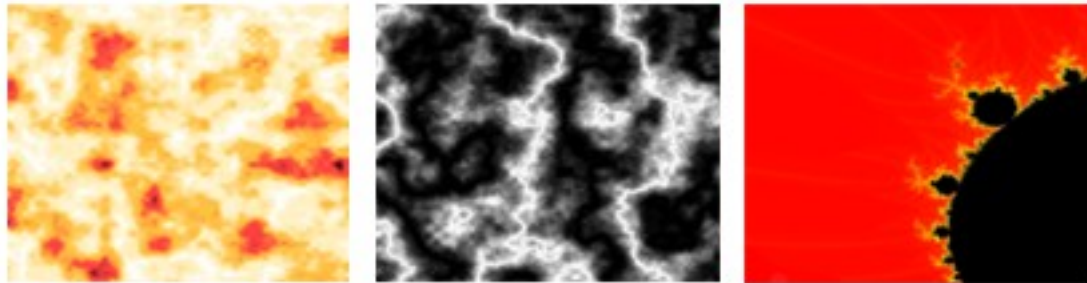


JNetic had a method of monitoring the best, average and worst of the population's fitnesses over generations. This feature is kept here, but stores the best and average of each fitness function (both raw and error) over generations.

In addition, the current population's statistics can be viewed, in a similar manner to the GP Tree Viewer. The user can choose the fitness function on both the X and Y axis, as well as the raw or error fitness to display. Each individual's fitnesses on the X and Y are displayed in a list below the graph.

Predefined Textures

JTexGen is a Java library that takes the ideas behind noise, waves, color mixing and other functions, and creates a set of predefined textures that can be generated with few parameters. These functions were



implemented in order to reduce the effort needed to find good textures, since these textures were already created to look good. These predefined textures can be mixed with one another or with other colors, and can be rendered as textured geometric primitives as well. There is a minimal slowdown with these implemented, which make them a good addition to the function set.

One current drawback, however, is that there are few of them implemented, and when crossover or mutation is applied, often there are not many to choose from so there will be little variability with small function sets. These work best without the texture mixing function.

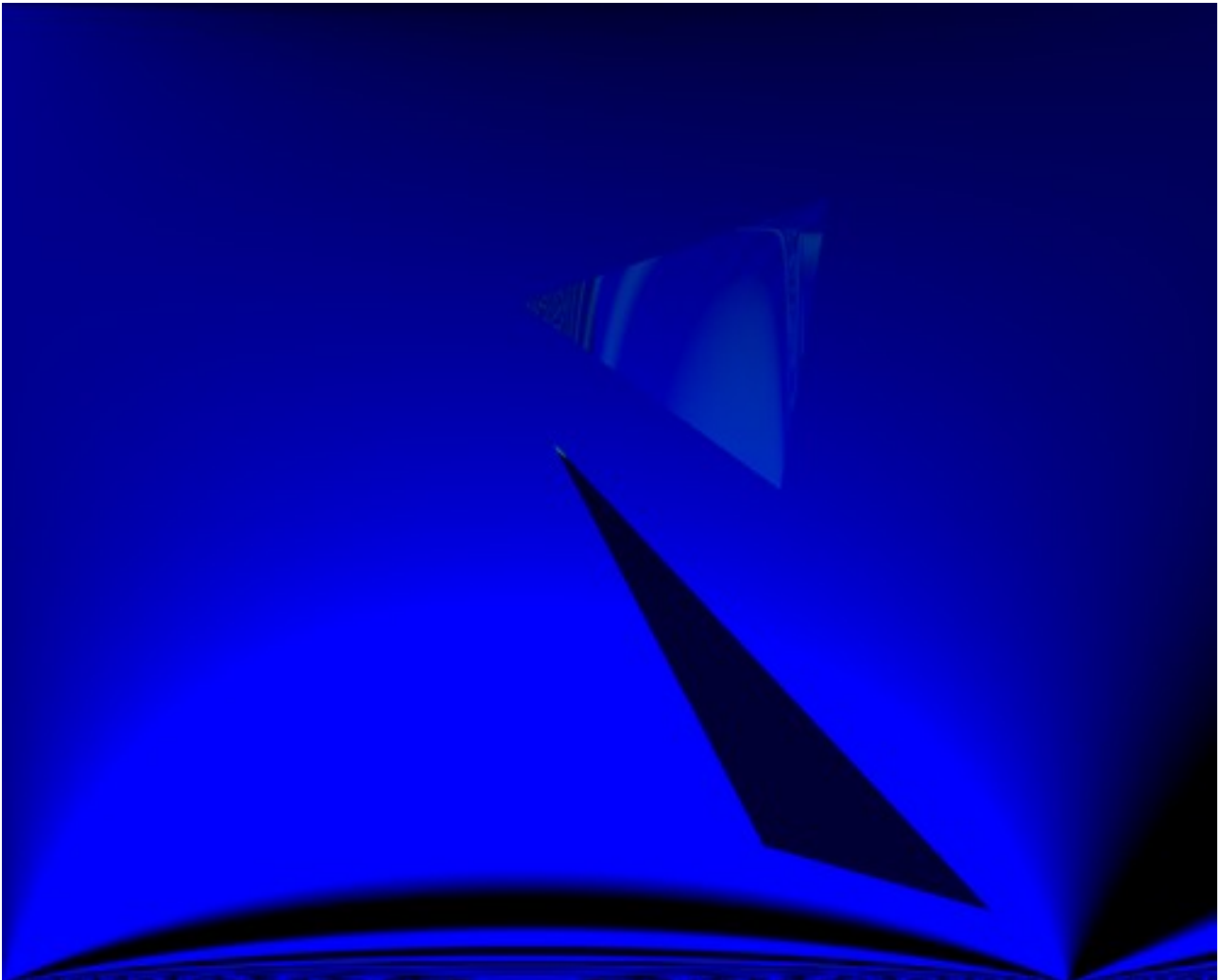
Textured Primitives

A feature uncommonly found in procedural texture generation is the concept of textured geometric primitives. It is rather surprising, considering the nature of procedural textures involve loss-less resizing; which is also a feature of geometric primitives.

JNetic textures adds three primitives as functions to its set; triangles, circles and rectangles. Each primitive can have its own texture, and a mixture of multiple primitives can be used in a single texture. Primitives also have the capacity to share the same texture via the use of ADFs.

Results

Reflection



MEAN : 2.899345984062473 (Goal of 3.0)

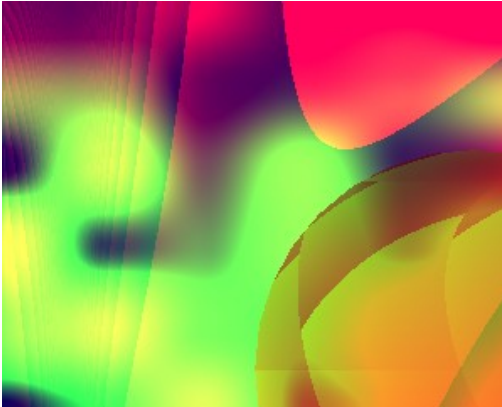
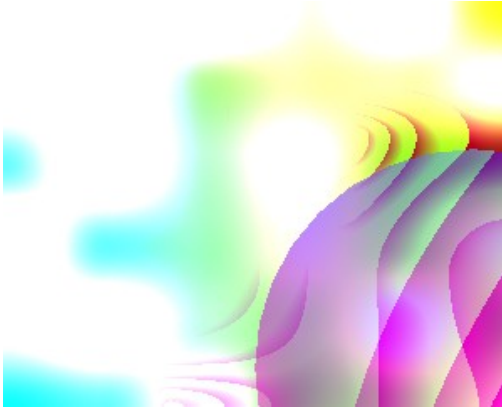
STD : 1.512821792019104 (Goal of 1.5)

DFN 1.751713547423095 (Goal of 1.75)

This image was an attempt to negate the evaluation of color, and instead focusing on the actual image and DFN. Personally, I find that the results produced by color histogram matching are not as interesting as those without. This is a perfect example of a texture that really isn't a texture. It almost looks as if there is a light source behind the viewer, and that there is reflection going on with the middle triangle. It's almost perfect. Also surprising is that there was no use of any ADFs in this image, yet both triangle and the background focus on a blue-and-black color scheme. I kept the function and terminal set simple, relying on all math functions except noise, and limiting the tree depth to 10.

This is my personal favorite.

Crystal Ball



| Error | Raw | Rank |
|------------------|----------|------|
| Mean : 0.004927 | 3.745073 | 18 |
| STD : 0.01099 | 0.76099 | 35 |
| DFN : 0.932829 | 0.932829 | 77 |
| CDist : 0.655225 | 0.655225 | 89 |

```
(rgb (noise ERCFloat[d4600375758924283904|
0.37215447425842285]]) (* (log (% (+ (noise
ERCFloat[d4597372442913013760|0.22771847248077393])) (sin u)) (-
(/ v (noise ERCFloat[d4599157298177245184|0.3045163154602051]))
(+ u (% u (/ u (sin u)))))) (power (- (/ (cos u) v) (cos (- u (noise
ERCFloat[d4601045769527492608|0.4093475341796875])))) (* (sin u)
(log (log v)))) u)
```

| Error | Raw | Rank |
|-----------------|----------|------|
| Mean : 0.110771 | 3.639229 | 237 |
| STD : 0.161313 | 0.911313 | 260 |
| DFN : 1.274832 | 1.274832 | 285 |
| CDist : 0.34623 | 0.34623 | 333 |

```
(rgb (cos (/ (- (* (noise ERCFloat[d4604550975552749568|
0.7078511118888855])) (% ERCFloat[d4600060368738320384|
0.35464680194854736]) ERCFloat[d4576836104031830016|
0.009857535362243652])) (cos (+ (% (cos v)
ERCFloat[d4601293124210262016|0.4230784773826599])) (cos (/ (log
ERCFloat[d4602990588164308992|0.534613311290741])) (cos v))))))
(log (power u ERCFloat[d4601026571023679488|
0.4082818031311035])))) (* (noise ERCFloat[d4604550975552749568|
0.7078511118888855])) (/ v (/ (sin (/ (- (noise
ERCFloat[d4596701528219189248|0.20909684896469116])) (+ (/ v (log
(log (noise ERCFloat[d4604350516007272448|
0.6855956315994263])))) (- (/ (% ERCFloat[d4599869641051865088|
0.3440592885017395])) (+ (noise ERCFloat[d4597372442913013760|
0.22771847248077393])) (sin u))) ERCFloat[d4603303416402280448|
0.5693442225456238])) v))) (* (noise
ERCFloat[d4605700785005658112|0.8355056047439575])) u))) (noise
ERCFloat[d4595590922690887680|0.1782713532447815])))) (cos u))
```

| Error | Raw | Rank |
|------------------|----------|------|
| Mean : 0.175243 | 3.925243 | 278 |
| STD : 0.220518 | 0.970518 | 297 |
| DFN : 0.975613 | 0.975613 | 164 |
| CDist : 0.462078 | 0.462078 | 303 |

```
(rgb u (cos (/ (sin (+ (sin (sin (sin
ERCFloat[d4598459863134109696|0.2658008933067322])))) u))
(- (noise ERCFloat[d4604758703256633344|
0.730913519859314])) (noise ERCFloat[d4600019376496705536|
0.3523712754249573])))) (cos v))
```

These three textures were taken from the population at generation 50. The population size used was 500, with a 100% crossover rate and 30% mutation rate. Grow was used as an initialization operator with depths 5-17.

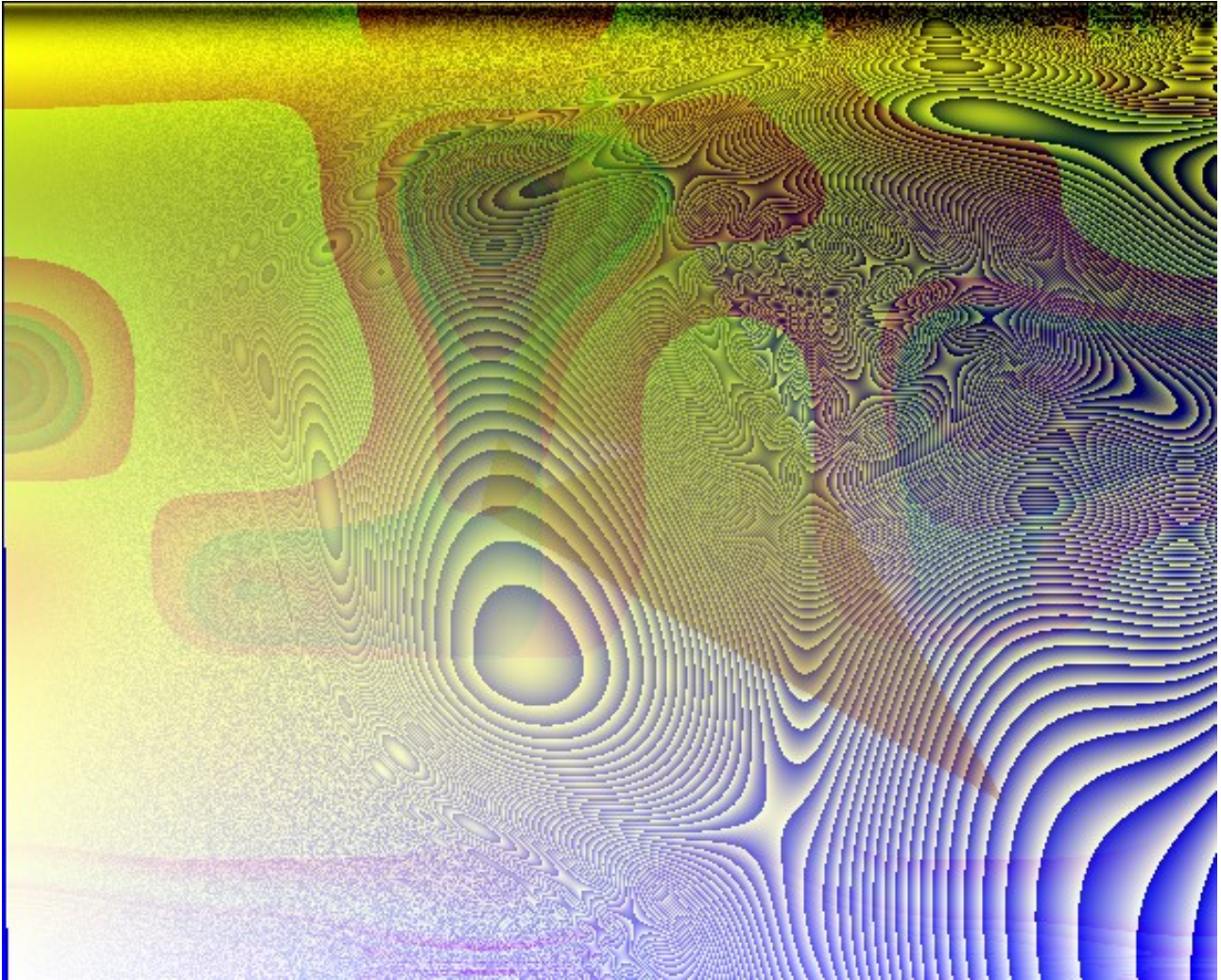
These are great examples to illustrate situations where the lower ranked images are often some of the best. The first image was the highest ranked in the population, and did fairly well compared to the others. The second image and the third were two of the worst images, but definitely exhibit interesting features. The texture on the circle in the second image almost makes it appear spherical, and the green 'blob' seems to glow. These are definitely affects you normally need to generate through trial-and-error in any other system. The addition of textured primitives is also a new addition to texture generation.



Source image

The noise function is perhaps the most powerful function in procedural texture creation, adding a random element to the mix. When combined with trigonometry functions, the results are truly fascinating. Color distance matching was using in this image in order to attempt to get as close as possible to the source. This is a very unrealistic goal here, but it definitely pushes the system to produce images as close as possible. In this case, the best image (with a color distance of 0.65) is actually fairly close, despite it resembles the source in no way. Color matching is suggested only for simple images, as in many cases the genetic programming system will always attempt to produce an image with colors with an average equal to the average RGB values of the source.

Differences



| | | | |
|---------|----------|----------|----|
| Mean : | 1.186389 | 5.313611 | 1 |
| STD : | 0.007716 | 0.757716 | 39 |
| DFN : | 0.078685 | 3.078685 | 34 |
| HDist : | 29890 | 29890 | 95 |

This is an image that uses the same source previously mentioned for fitness calculation. It also uses the same general GP parameters, but a different seed and histogram distance instead. It's goal mean, standard deviation and DFN are also different. This texture did fairly well with respect to mean, standard deviation and DFN, and was the highest ranked in the population. It is also a strange image; one that you might typically find through trial-and-error attempting to produce a good texture. It is not particularly interesting, but attains high scores. This only further proves the difficulty of finding an automatic image evaluation function.

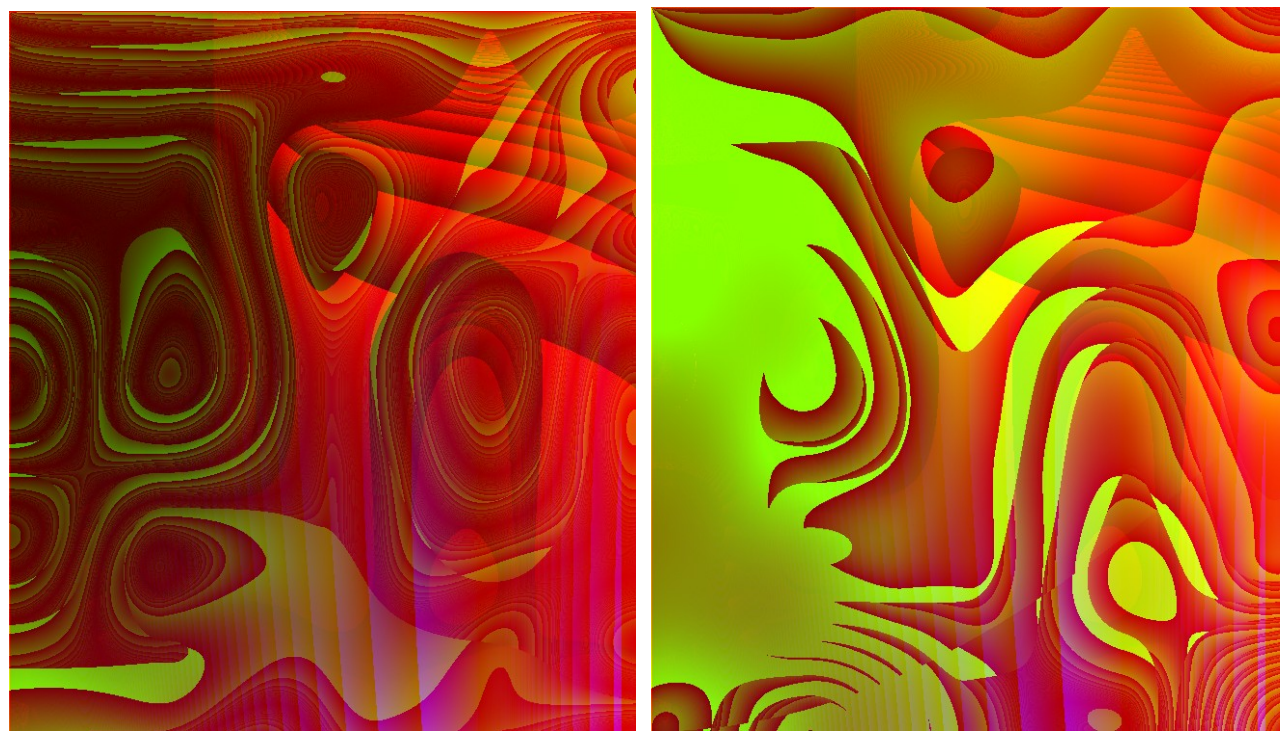
Cell



| | | | |
|---------|----------|----------|-----|
| Mean : | 2.733833 | 3.766167 | 81 |
| STD : | 1.200081 | 1.950081 | 251 |
| DFN : | 0.210991 | 3.210991 | 90 |
| HDist : | 34440 | 34440 | 188 |

This image was from the same population as above, but did not fare as well with respect to rank. This is another one of my favorites, and almost looks like something you might find under a microscope. The most interesting aspect of this image is how each 'cell' appears to be textured, and the combination of all cells share the same texture.

Eruption



| | | | | | |
|-----------------|----------|-----|-----------------|----------|-----|
| Mean : 0.159357 | 3.909357 | 131 | Mean : 0.053554 | 3.696446 | 53 |
| STD : 0.018041 | 0.731959 | 106 | STD : 0.742716 | 1.492716 | 386 |
| DFN : 0.313341 | 0.313341 | 49 | DFN : 1.380494 | 1.380494 | 359 |
| HDist : 89538 | 89538 | 141 | HDist : 96614 | 96614 | 309 |



Source image

These are two textures from generation 40 of this run, using similar parameters as above. These results are something without any primitives, utilizing a function set using two root functions; one with and one without primitives. The first was one of the more interesting results, being the second-highest ranked in the population. The second was one of the lower ranks. You can see the similarities between the two, especially looking at the textured background. This is a good example of small changes in the tree having large effects on the fitnesses.

The textured swirls and blobs are a combination of Perlin noise and trigonometry functions, which produce the most interesting results a large portion of the time. The 'layering' of textures is also something we see, which is also something that occurs explicitly when we add the 'Mix texture with RGB' function to the set. The user really has to experiment initially to get to know the function set, in order to visually understand how each function behaves. This is one of the better examples which shows how the addition of the color histogram distance function really has an impact on the palette used.

Conclusions

Future Work

Although I believe all of the primary and secondary goals of this project have been fulfilled, I believe that there is so much more that can be added to this project.

Additional/Extended Mutation and Crossover Operators

ECJ allows the user to specify the node selection probability for non-terminals and terminals, but an even better idea would be to give each function and terminal its own choice probability. If this were done, then it may solve the problems that came up regarding the JTexGen textures. Another idea would be to implement new mutation and crossover operators.

Additional Textures

I came across quite a few other procedural texture formulas, as well as some other ideas that would make good functions and terminals. The addition of more (or custom) texture functions would definitely be beneficial.

Chromosome Editing

Although this feature wouldn't be nearly as useful as it was in JNetic, the ability to change a tree's nodes at will would definitely be an interesting function, even if just for the experience of doing so. This would have been something I would have done in this project had time constraints not limited it.

Tree Loading/Saving

This was something I had really hoped to do by the end of the project, since it would be greatly beneficial for a user to save and load textures. ECJ has a facility to do this, but only into existing populations. In the future loading an existing texture into a population would be useful, but the existing population would have to use the *exact same function and terminal set and follow the exact same tree restrictions*. Another idea would be to load a tree and parameter file into a separate program (or window), just for the purpose of resizing and saving as an image. I saw this as a possibility for the future, and so I allowed the GP tree to be printed as a string as well as a tree-list.

Final Words

This was a very interesting project with a lot of possibility for future work. I gained a lot of experience working with genetic programming, as well as experience working deeply with two GP systems ECJ and JGAP. This was my first experience working with procedural textures as well, and it was an interesting topic. I also gained a lot of research experience, reading many papers from many sources, on many different topics.

It was great to be able to work on another project leaning towards image design and automated fitness calculation on subjective targets. It was such a different experience dealing with a fitness function that was so unlike direct color matching as well; it only further supports the idea that automated image generation is extremely difficult. This not only applied to images, but other media and design topics as well. Evolutionary computation applied to design still appeals to me greatly, and I hope to continue my work and thesis in this area.

Appendix

Creation Process

This section outlines my experiences with this project, as well as any other significant experiences I had with this reading course in general.

The initial project was to re-create what *Gentropy* was – an automated texture generation system – but also to incorporate it into a user-interactive environment in the same way Sean Wilkens' work was extended and integrated. This project would be different, however, as with JNetic the genetic algorithm system was custom-built, and genetic programming systems are much more difficult to build. Due to this fact, I needed to find a suitable Java-based GP system. I chose Java because of my experiences with it, and I am finding more and more that Java is not as slow as it used to be with respect to processing speed. I also planned on re-using JNetic's interface framework for this project as well, due to the similarities in image rendering with both projects.

The first system I investigated thoroughly was JGAP. This system was remarkably easy to get used to, as all parameters were easily-accessible and the documentation for frequently-used classes was readily available. It was this system that was first integrated into Jnetic's framework, and where most of the function and terminal set were implemented. Unfortunately, during the first real tests, the population converged heavily within the first few generations. With the lack of documentation for the operator classes, such as mutation and crossover, it was difficult to debug the actual cause of this issue. It turned out that mutation was never implemented at all, and crossover itself was questionable. Therefore, in the very worst case scenario, neither were being done and the best individuals were simply being replicated. I then decided to scrap JGAP for a more reliable system.

ECJ was the next system to try, and is the one now in implementation. For the most part, the system is highly reliable, though the learning curve is extremely steep. The use of parameter files is another setback, since all parameters are stored and read here only once, and trying to locate the parameters themselves as variables in existing classes is more difficult than it sounds. Due to the automatic threading and naming conventions in ECJ, often multiple instances of the same genetic operator classes exist during run-time, and it is near impossible to predict which one is currently being used. This fact led to some of the parameters being unchangeable during run-time, which is unfortunate.

Another setback was again the early convergence of the population. Despite the author's claims that mutation and crossover are present, further investigation showed that mutation was indeed *not* implemented, and crossover once again was questionable and frequently resulted in exact duplication of parents. The debugging process showed that everything else worked perfectly fine, so it was decided that I would stick with this system, implement mutation and fix crossover.

With mutation, I found an instance of sub-tree mutation that worked fine with few tweaks. It was a simple fix once I knew how to do so; the pipeline system in ECJ is used to direct the process from one operator to another, and is not documented well. Because the class was custom-built/edited, I had access to mutation rate and could change this at run-time.

The issue with crossover involved the algorithm that was implemented to find two compatible nodes; that is, two nodes with the same return type. The algorithm before editing was :

Choose 2 nodes randomly...

If nodes are compatible, switch sub-trees,

Otherwise, do nothing.

This class was not documented well, and this problem actually took a bit of time to find. The simple fix was to increase the number of times two nodes were picked randomly and tested. In all trees, at least some crossover should be possible, whether it be terminals or non-terminals. As with mutation, the crossover rate was also made available.

The original *Gentropy* system used a few functions that were omitted here, partly due to the fact that I wasn't sure what they did exactly. In their place, I looked for other functions to randomize numbers or create textures. In my search I came across Mandelbrot sets and JTexGen, a procedural texture generator. The idea behind JTexGen was that often procedural textures need tweaking, and instead they define a set of predefined textures with a small number of parameters. In this way they are actually reducing the search needed for interesting textures, and so some of these were added to the function set.

The results were interesting, but posed a few problems. Due to the limited number of these textures and the fact that there are only two rgb functions (one with textures, one with doubles), during crossover and mutation these textures are often chosen more than the rgb-generated textures. These textures usually already have good DFN scores as well, and so usually rank the highest and push out the rgb-generated textures early on. In the future, I plan on finding a way to remedy this.

Another aspect I actually spent a good deal of time researching was how to view a GP individual. Discussions with Professor Ross about how to graph a population with respect to 2 different fitness functions inspired the idea that, while I'm at it, I might as well pinpoint specific individuals as well in this graph. Initially, the actual GP tree was viewable by a string alone, which is both hard to read and visually unappealing. I came across the idea to use Java-tree-lists from Netbeans, as I was commenting to myself about how much easier it is to use an IDE to program large-scale projects due to the file-organization system. When I was looking around for other systems that did the same, I could find few. This is something I hope will catch on since it made debugging much easier, especially when I had to debug crossover and mutation.

Experiments were done with another fitness function, direct color distance matching. This was the same fitness function used in JNetic. The results using this method were not too impressive, as most textures attempted to average the colors in the source image in order to get the highest fitness possible. What resulted was a lot of white, black, gray and brown images. This function on its own doesn't work too well, but may work better accompanied by others.

In addition to the research material I read for this project, I also read a good amount on genetic programming applied to shape grammars and structure design for my thesis. I decided to do this for two reasons; because this project focused on genetic programming and genetic programming when applied to design is a large area of research. Some of the more interesting papers I read I included in the literature search.

Literature Search

Bergen, S. R. 2009. Evolving stylized images using a user-interactive genetic algorithm. In Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers (Montreal, Québec, Canada, July 08 - 12, 2009). GECCO '09. ACM, New York, NY, 2745-2752.

I thought it was good to reference this paper, as I am re-using the interface, as well as some menu and image-parsing functions.

Bhat, P., Ingram, S., and Turk, G. 2004. Geometric texture synthesis by example. In Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing (Nice, France, July 08 - 10, 2004). SGP '04, vol. 71. ACM, New York, NY, 41-44.

This paper introduced me to the concept of '3D texturing', without the use of 2D images. These 3D textures are geometry-based and are procedural in nature. The results of the application of these volumetric textures is interesting, and actually very similar (though less complex) to the idea proposed in (Kopf, 2007). This process seems to be an undesired alternative to bump-mapping due to rendering complexity.

Brooks, S. and Dodgson, N. A. 2005. Integrating procedural textures with replicated image editing. In Proceedings of the 3rd international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia (Dunedin, New Zealand, November 29 - December 02, 2005). GRAPHITE '05. ACM, New York, NY, 277-280.

Introduces a system to automatically generate textures with minimal user interaction. The idea is to allow the user to decide the changes they want to make to an image (likely involving color or sectional image tranformation), and apply the same changes across the whole image automatically. The idea is a little simple, but some of the results are actually impressive. This serves more as a filtering technique than a texture generator.

Cook, R. L. and DeRose, T. 2005. Wavelet noise. In ACM SIGGRAPH 2005 Papers (Los Angeles, California, July 31 - August 04, 2005). J. Marks, Ed. SIGGRAPH '05. ACM, New York, NY, 803-811.

Paper describes in detail extensions of Perlin's noise function in an attempt to improve upon areas of error with aliasing and some loss of detail. The use of wavelet theory is applied to the generalized noise function, producing interesting results. I looked into this paper for a better representation of a noise function, but the added complexity using wavelets turned me off of the idea at this junction. Future work might bring me back to this idea.

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. 2002 Texturing and Modeling: a Procedural Approach. 3rd. Morgan Kaufmann Publishers Inc.

This book contains several procedural texture rendering algorithms, and discusses a great deal of preliminary information which I found useful in early stages. Since I focused more on getting the GP system working with a few textures, I was not able to put this book to its full use. Future work will likely bring me back to read further, as a variety of textures are available as well as explanations on their use and extension.

Gibson, A. *JTexGen - Procedural Texture Library*.

<http://www.andygibson.net/blog/index.php/2009/08/02/jtexgen-procedural-texture-library-released/>

The idea to use JTexGen came from looking at the images in a few other procedural texture papers. This library contains pre-written algorithms which generate procedural textures from parameters. I decided to replace some of the older functions (clouds, warp, etc.) with these new textures, to see if perhaps it would decrease the search space. Unfortunately, time constraints limited the number of textures implemented, which had an effect on results.

Horn, J., Nafpliotis, N. and Goldberg, D. E. *A niched Pareto genetic algorithm for multiobjective optimization, in Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence (Cat. No.94TH0650-2), Orlando, FL, USA, 1994, pp. 82-7.*

This was one of the first papers I had read regarding Pareto-ranking in multiobjective systems. My initial implementation of pareto ranking was based on the method described here (though extended for GP).

Johanson, B. and Poli, R. *GP-Music: An Interactive Genetic Programming System for Music Generation with Automated Fitness Raters*. 1998.

This paper describes an interactive GP system that evolves short musical sequences. The system monitors a population of trees containing terminals, which are notes, and functions, which edit the note set. Fitness evaluation is initially interactive, requiring the user to evaluate each individual. A neural network steps in later to evaluate further populations based upon the rating system the user previously followed. The idea that a neural network could be implemented to monitor the user and step in later is a novel idea.

Kopf, J., Fu, C., Cohen-Or, D., Deussen, O., Lischinski, D., and Wong, T. 2007. *Solid texture synthesis from 2D exemplars*. In *ACM SIGGRAPH 2007 Papers (San Diego, California, August 05 - 09, 2007)*. SIGGRAPH '07. ACM, New York, NY, 2.

This paper describes a process of taking 2D texture 'exemplars', and applying them to 3D models. The key difference here is the introduction of different channels for each exemplar texture, such as shininess, color, displacement, etc. Each channel has a different affect on the 3D model, producing fantastic results. The process itself is complex, and likely not to be used in real-time applications anytime soon. This paper caught my eye originally as I was looking for a 2D simple bump-mapping function. Though it does not apply to this project, I still enjoyed the read.

Koza, J. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

I read only small areas of this book, though felt it necessary to add it to this list. This is actually where I learned about the application of symbolic regression problems to GP, which I now use to test my problem implementations in systems prior to larger problems. This book, as one of the early books on the topic of GP, is highly theory-based. It is of interest to me to look into the newer versions in the future.

Luke, S. *ECJ - A Java-based Evolutionary Computation Research System.*

<http://cs.gmu.edu/~eclab/projects/ecj/>

ECJ is a Java-based system with a primary focus on GP. It bases most of its architecture on that of lilgp, with the GP portions taken right from Koza's work. The system has a hefty learning curve, mostly due to lack of documentation and cryptic parameter sets. The extension of existing classes is also painstakingly difficult at first. Once you get used to this system it is remarkably simple, albeit tedious, to apply it to any problem. The multi-objective facilities are still at their early stages, and are not very dependable. This, along with the mutation and tournament selector, needed to be altered greatly for use in this project.

Neufeld, C. Ross, B. J., Ralph, W. *The Evolution of Artistic Filters. In The Art of Artificial Evolution, P. Machado and J. Cardalda (eds.), Springer, 2007.*

This was one of the first papers I read. It didn't give me much information regarding texture evolution, but it did introduce me to Ralph's model of aesthetics. It also serves as a good example of image evolution.

O'Neill, M., Swafford, J. M., McDermott, J., Byrne, J., Brabazon, A., Shotton, E., McNally, C., and Hemberg, M. 2009. *Shape grammars and grammatical evolution for evolutionary design. In Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (Montreal, Québec, Canada, July 08 - 12, 2009). GECCO '09. ACM, New York, NY, 1035-1042.*

I read this paper initially looking for information on shape grammars with genetic programming. In this case, the grammar used is a simple one which involves translations and scaling of geometric primitives. The purpose of the paper was to generate images close to the original (similar to JNetic) in order to prove the value of shape grammars. It's easy to see that the idea of shape grammars can be applied to any grammar (something like Maya or CityEngine).

Özkar, M. and Stiny, G. 2009. *Shape grammars. In ACM SIGGRAPH 2009 Courses (New Orleans, Louisiana, August 03 - 07, 2009). SIGGRAPH '09. ACM, New York, NY, 1-176.*

This paper focuses on the basics of shape grammars and the history and applications of them. It was an interesting read (it's actually a tutorial), as it describes how shape grammars are more commonly used in genetic algorithms in order to abstract the problem. In this way, implementation is difficult but in the end one implementation is made, it can be extended to any grammar language.

Parish, Y. I. and Müller, P. 2001. *Procedural modeling of cities. In Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '01. ACM, New York, NY, 301-308.*

This paper describes a procedural building modeling system, CityEngine. I read through this paper during this project in order to potentially apply some of their procedural generation techniques (which base themselves on a 2D plane of road-map information) to a textured image. Although I did not find anything useful to that extent, I do find the idea interesting, and the paper itself might prove useful for my thesis as it describes in better detail the processes behind the city generation.

Perlin, K. 2002. *Improving noise. In Proceedings of the 29th Annual Conference on Computer*

Graphics and interactive Techniques (San Antonio, Texas, July 23 - 26, 2002). SIGGRAPH '02. ACM, New York, NY, 681-682.

This paper describes the differences between Ken Perlin's old and improved noise function, as well as the implementation aspects of the new. This paper was found looking for an implementation of the improved noise algorithm, and was the only publication I could find by Perlin on the subject.

Poli, R. and Langdon, W. B. 2009. *Genetic programming theory I & II. In Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers* (Montreal, Québec, Canada, July 08 - 12, 2009). GECCO '09. ACM, New York, NY, 3015-3056.

This was a tutorial I attended at GECCO '09, but the accompanied text was easier to follow on my own time. It describes more background information relating to GP and the search space. At the time of this reading, I knew only the basics behind the idea of search space. This paper/tutorial focuses more heavily on theory than implementation, and so I took little from this the first time I read it.

Poli, R., Langdon, W. B., McPhee, N. F. *A Field Guide to Geneting Programming*. www.gp-field-guide.org.uk.

This book was by far the most useful when I began this project. It introduced me to the key features of genetic programming in a step-wise manner, focusing on the basics early on and introducing more advanced features later. I returned to this book countless times, especially when I switched from JGAP back to ECJ, in order to better understand some of the more advanced topics (such as ADF's).

Qu, B. Y. and Suganthan, P. N. 2009. *Multi-objective Evolutionary Programming without Non-domination Sorting is up to Twenty Times Faster*. Special Session on Performance Assessment of Multiobjective Optimization Algorithms/CEC 09 MOEA Competition. Norway.

This paper is the closest reference I could find for rank-sum multiobjective scoring. The algorithm they describe is exactly the same as the one I am now using for this project. They do not reference anyone else for this algorithm, either.

Ross, B. J., Ralph, W. and Zong, H. *Evolutionary Image Synthesis Using a Model of Aesthetics*. Conference on Evolutionary Computation (CEC 2006), Vancouver, BC, July 2006.

This paper describes the methods behind the DFN scoring algorithm. I also used this to set up my function and terminal sets. A few of the functions - such as clouds and warp - I did not implement. Instead, I replaced them with automatically generated textures using jtexgen. Most of the GP parameters that I used - such as population size - I got from here as well.

Rotstan, N. *JGAP - Java Genetic Algorithms Package*. <http://jgap.sourceforge.net/>

This system works with both GA and GP, and at first was simple to get something up and running. Unfortunately, extensive use showed that no mutation operator existed, and multiobjective facilities did not exist. The extension of existing classes to facilitate custom functions was extremely difficult, as the system is

hard-coded to work a specific way and very rigid. It does serve as a good start for beginners to GP and GA, and for simple problems.

Sun Microsystems. 2002. *Project Swing*. <http://java.sun.com/j2se/1.5.0/docs/guide/swing/index.html>

Once again, I felt it was necessary to include this as I worked extensively with Java and Swing.

Talukder, A. A., Kirley, M., and Buyya, R. 2008. *A pareto following variation operator for fast-converging multiobjective evolutionary algorithms*. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (Atlanta, GA, USA, July 12 - 16, 2008)*. M. Keijzer, Ed. GECCO '08. ACM, New York, NY, 721-728.

I came across this paper when I was searching for an alternative to pareto ranking. My implementation of pareto ranking seemed to converge too quickly, even with massive population sizes. This paper proposes a new operator which improves the existing pareto technique with respect to early convergence. Two search spaces are monitored 'objective and design variable space'. Each pareto front is chosen not only based upon non-domination, but from a dynamic function of the two spaces (dynamic since the design space will constantly change). This method seemed to do quite well, but the overall complexity of the implementation seemed excessive for the scope of this project, so I decided to seek other means.

Wiens, A. L. and Ross, B. J. *Gentropy: Evolving 2D Textures, Computers and Graphics*, v.26, n.1, Feb 2002, pp. 75–88.

This paper served as an extremely useful resource for the algorithms and formulae behind the color histogram method of fitness calculation, as well as the introduction of other methods. This is also where the color histogram quadratic matching formula is found, which I tested extensively and found it far too computationally expensive to use on large images. Instead I replaced it with a simplified version based on its method.

Xu, Q., D'Souza, D., and Ciesielski, V. 2007. *Evolving images for entertainment*. In *Proceedings of the 4th Australasian Conference on Interactive Entertainment (Melbourne, Australia, December 03 - 05, 2007)*. ACM International Conference Proceeding Series, vol. 305. RMIT University, Melbourne, Australia, 1-8.

This was one paper I read initially when working on JNetic. It introduced IMAGENE, a procedural texture generation system very similar to Gentropy. I looked back at this paper in the early stages of the project to get the basic ideas, as it is far more simple than Gentropy with respect to its function and terminal sets.

User Manual

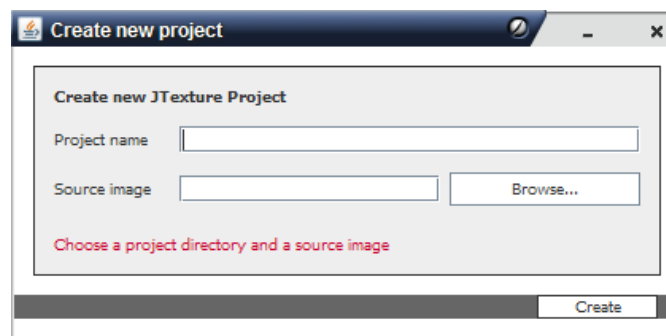
This user manual will guide you through the controls and windows of JNetic Textures. This manual should be used in conjunction with the paper on this topic, so that you can get a better understanding behind genetic programming, the system, and the fitness functions used.

Getting Started

The first thing that needs to be done in order to use JNetic Textures is to double-click the JAR file, or run the JAR file from the command line with

```
java -jar jarname.jar
```

Once this is done, you'll be prompted with this window :



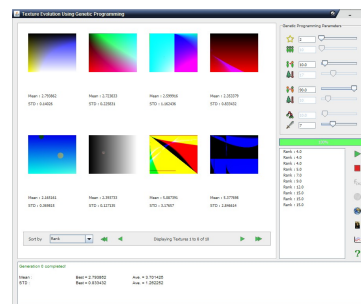
Define a project name, which can be anything. Note that when a project is created, a folder is assigned with the project name to the 'Projects/' folder. This is where all your parameters are stored.

Next, define a source image using the browse button. A source image must always be chosen, but is not always used (*see fitness functions for more details*). Even if the source image is not used for fitness calculation, it must still be chosen as a safeguard.

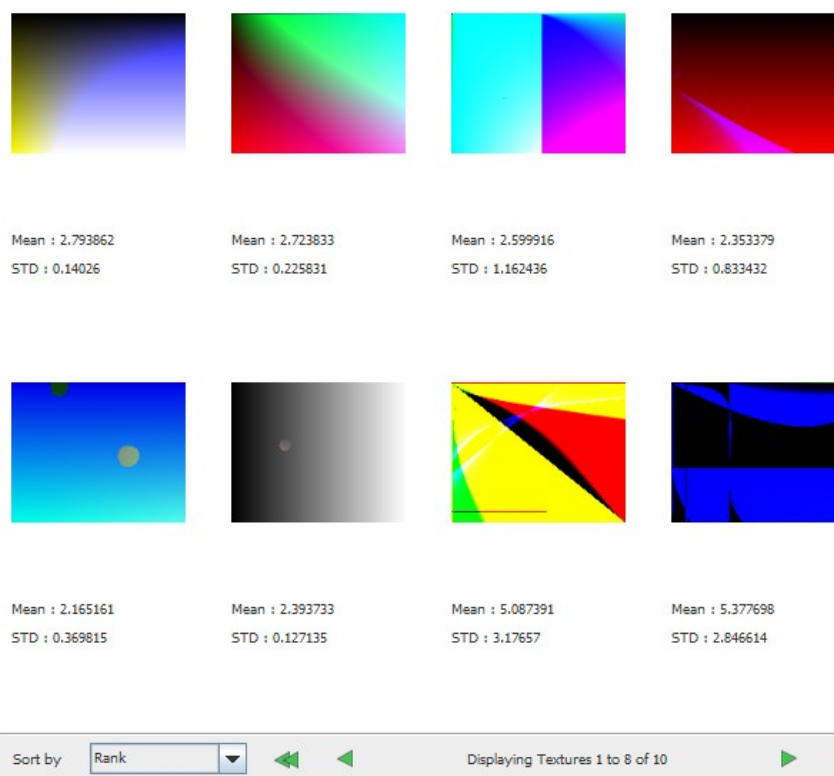
Main Window

The main window displays all the current progress of the run, as well as giving the user a good amount of control. This window stays open during the entire course of the run, and if closed will completely exit JNetic Textures. Despite the amount of user control, this is an automatic texture generation system and once the play button has been pressed, the genetic programming run will continue as long as the window is open.

This window is split into sections :



Tree Display



The tree display is used to show eight GP trees of the current population at a time. This window is updated once per-generation, after fitness evaluation has taken place, and displays the top eight ranked trees of the population. To change which trees are displayed, you can click on one of the four green arrows in the menu, which, in order from left-to-right :

- Move to the first page (top 8 textures)
- Move to the previous page
- Move to the next page
- Move to the last page

JNetic Textures

You can also change how each texture is sorted within the viewer with the drop-down box. By default, they are sorted by rank, but you can sort them by any of the fitness functions assigned as well. Once a sorting function is chosen, it often takes time to re-sort and re-display everything, so be patient. Also, once a sorting function is chosen, it is used throughout the rest of the run until changed.

The texture images can also be double- and right-clicked. By double-clicking an image, the *GP Tree Viewer* window appears. Right-clicking gives you the choice of displaying the image at its full dimensions, or opening the viewer.

Update Panel

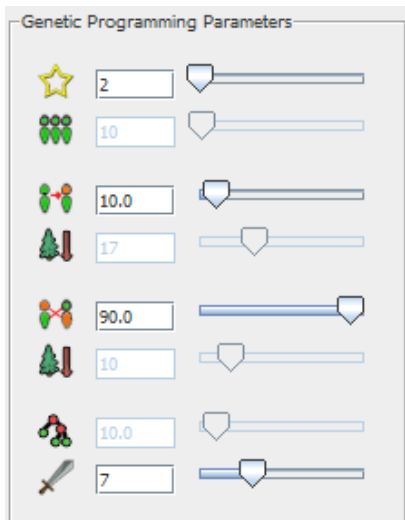
Generation 0 completed!

| | | |
|--------|-----------------|-----------------|
| Mean : | Best = 2.793862 | Ave. = 3.701425 |
| STD : | Best = 0.833432 | Ave. = 1.262252 |

This text area is used to update you on:

- The current generation
- The best and average of each fitness function with respect to the population

Genetic Programming Parameters

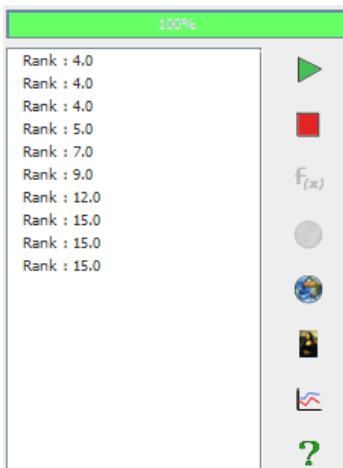


The GP Parameters panel offers a few primary controls for the GP run. Below is a list of all the parameters from top-to-bottom and their purpose, as well as if they can be changed during run-time, or can only be changed prior to run-time.

1. **Generation span (change any time)** : This sets the maximum amount of generations to elapse before the GP run quits. If this is changed to less than the current generation during run-time, the run will quit after the current generation completes.
2. **Population size (pre-run param)** : This sets the population size (the number of textures to produce).
3. **Mutation rate (change any time)** : This changes the rate of mutation. A 100% mutation rate will mutate every GP individual, where a 0% rate will never mutate anything.
4. **Mutate Maximum Depth (pre-run param)** : This defines how far down the tree to look for a node to be mutated. This is set in order to limit the size of the tree, since if each terminal has the capacity to grow by 5 every generation by mutation, then the tree will grow very quickly.
5. **Crossover Rate (change any time)** : This defines the rate of crossover, and acts in the same way as mutation.
6. **Crossover Maximum Depth (pre-run param)** : This defines how far down the tree to look for a node to crossover. As in mutation, it is used to limit the tree size.

7. **Choose-Terminal Probability (pre-run param)** : Defines the chance of choosing a terminal over a non-terminal in mutation and crossover.
8. **Tournament Size (change any time)** : This is the size of the tournament during selection, and defines how many individuals are picked for comparison.

Progress and Menu



This panel is split into three sections. At the top is the progress bar, which displays how far into the current generation the run is. This includes reproduction and evaluation.

At the left is a list of all the individuals in the current population, sorted by rank. This exists to show the user how many of each rank is present, and if perhaps the crossover or mutation rate needs to change.

The menu contains several functions for parameter changing and management. The button functionality is as follows, from top-to-bottom :

1. **Play/Pause** : This button starts the GP run, and is used to resume a paused GP run. Once this is pressed for the first time, a new parameter file is created and saved to the project directory, and the population is initialized. Also, once it is pressed, the play button turns into the pause button. When the pause button is pressed, the GP run halts once it has completed its current generation, and some parameters can be accessed once more.
2. **Stop** : The stop button can only be pressed once the GP run is paused. Once pressed, all controls are enabled once more and all current progress is lost and cannot be continued. You can still view and save the current population, however.
3. **Fitness Functions** : This opens the *Fitness Functions* window, allowing the user to select up to four fitness functions to be used in fitness calculation. This button is disabled at run-time.
4. **Function/Terminal Set** : This opens the *Function and Terminal Set* window, allowing the user to select from a large number of functions and terminals. This button is disabled at run-time.
5. **Globals** : This opens the *Globals* window, giving access to most of the system and some GP parameters. This button is enabled at all times.
6. **Source Image Display** : This opens the *Source Image* window, displaying the source image chosen. This window can be kept open at all times.
7. **Progress Chart** : This opens the *Progress Chart* window, which displays the population's progress.
8. **Help Button** : Opens the help file.

Fitness Functions Window

| Fitness Functions | Comparison Value |
|---|--------------------------|
| <input checked="" type="checkbox"/> Mean (reduce error) | 3.75 |
| <input checked="" type="checkbox"/> Standard Deviation (reduce error) | 0.75 |
| <input type="checkbox"/> DFN (deviation from normal) | 0.0 |
| <input type="checkbox"/> Direct Color Distance Matching | compared to source image |
| <input type="checkbox"/> Color Histogram Distance Matching | compared to source image |

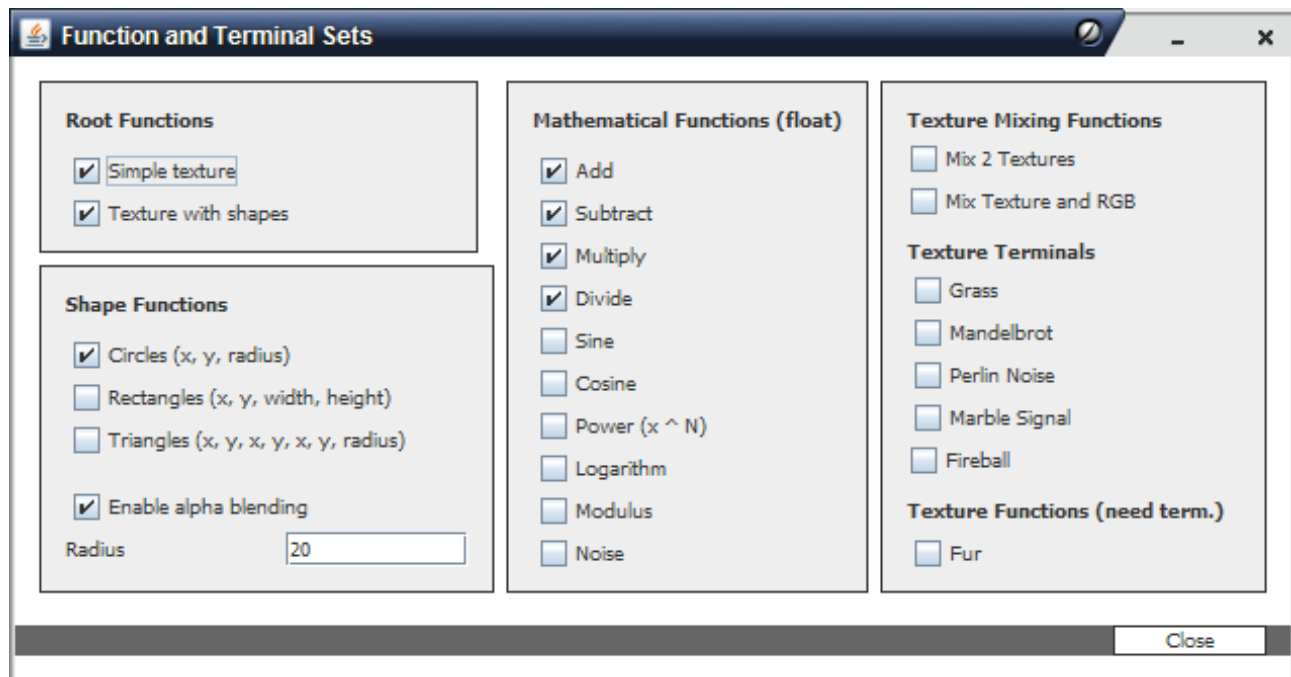
This window allows the user to select between 1 and 4 of the available fitness function for use in ranking. The first three fitness functions allow you to supply a comparison value, which is used in computing the error for each of these functions. For example, if we select **Mean** as one function and set the comparison value to 3.75, if one individual's mean value is 2.5, then its fitness is the error between the two values, which is 1.25.

The other two fitness functions depend on a source image. In direct color distance matching, the source image and the

texture image **MUST** be the same dimensions, or an error will occur. Therefore, in the *Globals* window, make sure the image height and width equal that of the source image. In color histogram distance matching, the sizes of the two images can differ.

For more information on the fitness calculations themselves, see the section on *Fitness Functions*.

Function/Terminal Set Window



The 'Function and Terminal Sets' window is divided into three main sections:

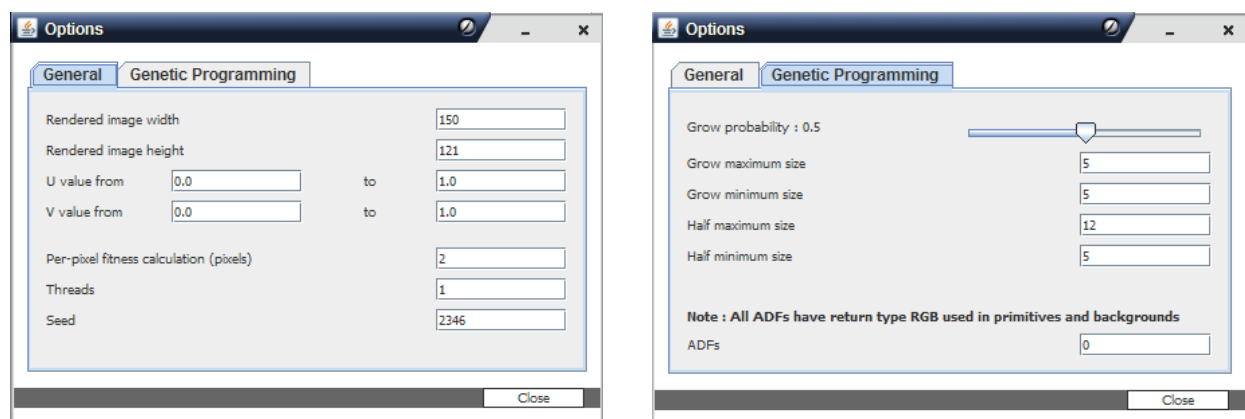
- Root Functions:**
 - ☒ Simple texture
 - ☒ Texture with shapes
- Shape Functions:**
 - ☒ Circles (x, y, radius)
 - ☐ Rectangles (x, y, width, height)
 - ☐ Triangles (x, y, x, y, x, y, radius)
 - ☒ Enable alpha blending
 - Radius:
- Mathematical Functions (float):**
 - ☒ Add
 - ☒ Subtract
 - ☒ Multiply
 - ☒ Divide
 - ☐ Sine
 - ☐ Cosine
 - ☐ Power (x^N)
 - ☐ Logarithm
 - ☐ Modulus
 - ☐ Noise
- Texture Mixing Functions:**
 - ☐ Mix 2 Textures
 - ☐ Mix Texture and RGB
- Texture Terminals:**
 - ☐ Grass
 - ☐ Mandelbrot
 - ☐ Perlin Noise
 - ☐ Marble Signal
 - ☐ Fireball
- Texture Functions (need term.):**
 - ☐ Fur

A 'Close' button is located at the bottom right.

This window allows you to choose between the many functions and terminals. You can select any combination of these, but there are a few things to note :

1. When choosing a root function, make sure that if *Texture with shapes* is chosen, that at least one of the shape functions are chosen below.
2. When choosing either *Mix 2 Textures* or *Mix Texture and RGB*, remember to choose at least one of the texture terminals.
3. When choosing *Fur*, remember to choose at least one texture terminal.

Globals Window



The 'Options' window has two tabs: 'General' and 'Genetic Programming'.

General Tab:

- Rendered image width:
- Rendered image height:
- U value from: to:
- V value from: to:
- Per-pixel fitness calculation (pixels):
- Threads:
- Seed:

Genetic Programming Tab:

- Grow probability : 0.5 (slider)
- Grow maximum size:
- Grow minimum size:
- Half maximum size:
- Half minimum size:
- Note : All ADFs have return type RGB used in primitives and backgrounds
- ADFs:

Both tabs have a 'Close' button at the bottom right.

The globals window provides the user with two panels of parameters to choose from, each with their own

JNetic Textures

parameter set :

General

Image width : The width of the rendered textures. It is a good idea to keep this the same as the source image if *Direct Color Matching* is one of the fitness functions chosen.

Image height : The height of the rendered textures. It is a good idea to keep this the same as the source image if *Direct Color Matching* is one of the fitness functions chosen.

U value range : The range of U values to use. These can be set to any values, and need not be the same as the V range.

V value range : The range of V values to use. These can be set to any values, and need not be the same as the U range.

Per-pixel calculation : Sets the number of pixels (- 1) to skip when performing *Direct Color Matching*. This is used to speed up calculation.

Threads : The number of threads to use in fitness calculation.

Seed : The random seed value used (cannot be changed during run-time!).

Genetic Programming

Grow Probability : The probability that *Grow* is chosen in *ramped half-and-half*.

Grow Maximum/Minimum : The range of tree depths to fill in *Grow*.

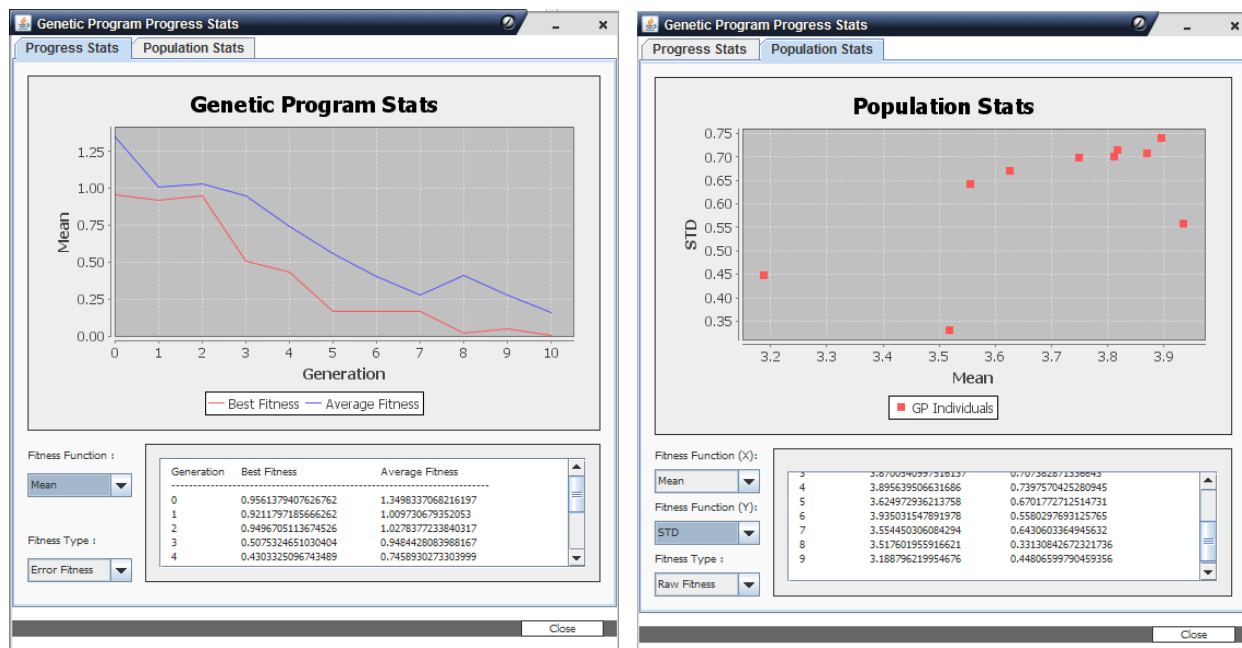
Full Maximum/Minimum : The range of tree depths to limit in *Half*.

ADFs : The number of Automatically Defined Functions to use.

Source Image Window

This window displays the source image at its full dimensions, and can be kept open at all times. This window also doubles as a population viewer when opened by right-clicking a texture in the *Tree Display*, and selecting *View Full Image*. Once opened in this way, you can use the navigation arrows to view each texture one-by-one.

Progress Chart Window



Progress Stats

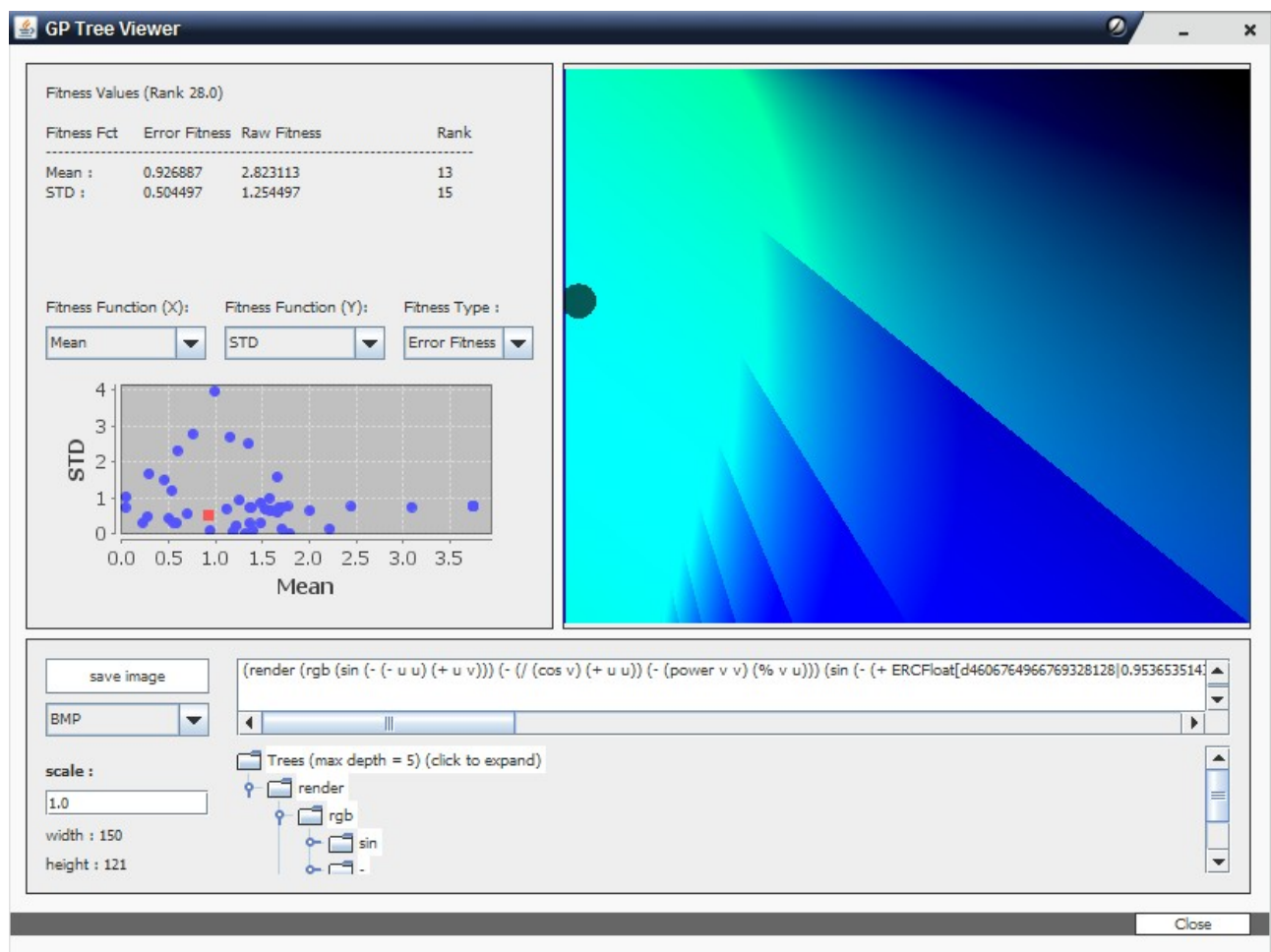
This panel displays the progress of the best and average fitnesses of each generation, for every fitness value selected. The X axis always consists of the number of generations, and the Y axis can be changed by selecting the fitness function in the drop-down box. You can also select the fitness type, raw or error, in order to display either the actual fitness value, or the error between the actual and the comparison value.

In the list below the chart is the actual values that are graphed.

Population Stats

In this panel, the current population is graphed according to two fitness functions, which can be chosen with the two drop-down boxes for the X and Y axis. Just as in the *Progress Stats* panel, you can also choose the fitness type. In the list below the chart are the values graphed, along with the ID of the individuals.

Tree Viewer Window



The Tree Viewer allows the user to monitor the progress of an individual in the population. It displays the texture at its full dimensions at the right. At the top-left, there is information about the fitness values of this individual, such as its rank-sum score, its fitnesses and their ranks within the population. There is also a graph similar to that in the *Progress Chart* window, allowing the user to select the X and Y axis fitness functions, and displaying this individual's place with respect to the rest of the population.

At the bottom left is an image-saving utility. The user can select the type of image and the scale of the new image (which, when changed, will show the new width and height below).

The bottom right displays the tree in two forms; in a textual format which can be copied out as a string, and a Java-tree-list format. In the list format, non-terminal nodes are represented by folders and can be opened revealing any children. This is the preferred method to viewing the tree.