# CS 351 Fall 2014, Project 2

`TriangleGenome`

Parallel Genetic Algorithm for Image Compression

**STATUS** Specification and requirements document

**VERSION** 1.0

**DATE** Sept 20, 2014

## 0 Changlog

**Version 1.0** Initial release. Sept 20, 2014.

## 1 Summary

The great minds of Museware (your employer) have decided to leverage the great minds of students in CS-351 to develop an image compression algorithm that uses as its underlying structure an ordered set of 200 semi-transparent, overlapping triangles. Each triangle is parameterized by 10 values: The pixel location of its three vertices relative to the upper left corner of the image ($x_1$, $y_1$, $x_2$, $y_2$, $x_3$, $y_3$), the triangle's color ($r$, $g$, $b$) and the triangle's transparency, ($a$). If a sequence of 200 triangles can be found that gives a "good enough" approximation of an image, then significant compression will have been achieved. For example, an uncompressed 24-bit color, 256×256 pixel image is about 200 KB of information. By contrast, 200 triangles with 10 bytes of data each, is only 2 KB. The definition of "good enough" is application dependent. It is very unlikely that this compression method will ever be used in a digital camera. However, it may be useful for transmitting a video communication across the solar system. That is, an application where 1) bandwidth is very expensive, 2) high visual fidelity is not important, and 3) the speed of light already prevents the communication from being interactive, and therefore, realtime compression is not required.

The tricky part, of course, is finding the "right" set of overlapping, semitransparent triangles and the "right" order in which to draw them. This will be accomplished by using a combination of both *hill climbing* and *genetic algorithms*. Furthermore, since these algorithms tend to be computationally intensive, it is important that they leverage the full potential of the available hardware (a.k.a. "leave no core with idle cycles").

Having overseen the development of other genetic algorithms, Museware knows that such algorithms tend to be rather more "sensitive" to design choices than more analytic approaches. As a result of this sensitivity, genetic algorithms, often require many iterations of redesign. This fact, coupled with the knowledge that at any time Google or Facebook might hire away the CS-351 class, Museware requires all of its developers to use the Git version control system. Git will document the history of redesigns so that when a single-person development team moves to bigger and better things, her replacement can examine the history of dead-end exploration rather than rediscovering the trails.

Another mitigating practice Museware enforces on its developers of performance and design sensitive algorithms is the collection and statistical analysis of quantitative timing data. Finally, as with all quality development efforts, Museware requires its developers to design and carry out unit testing with Java's `assert` keyword (not to be confused with `org.junit.Assert` which is a very fine unit testing method, but not the Museware standard).

## 2 Group/Individual Effort

This project is a group programming project. Each group is responsible for subdividing the project work into reasonably distinct parts with a single member assigned to each part. The different parts will be graded separately so that different members in the group may get widely different grades. However, it is the full group's responsibility to see that all the parts are working together and thus, no person in a group can get an "A" is the program as a whole does not function.

## 3 Java Language Level

Developers must write this project in Java 1.7.x (a.k.a., "the Java 7 platform") or a later version. Your code must support generic classes where required by the required project interfaces.

## 4 Libraries and Support Code

Developers may use any objects or functions available through the Java 1.8.x JDK. Developers MUST NOT use any other libraries or code, beyond those standard with Java 1.8, without prior permission from the CS-351 instructors.

## 5 Requirements

This section describes the elements that must be developed as part of this project. The designer may also choose to implement additional Java source files, programs, and/or shell scripts in support of the following items.

### 5-1 Coding Standard

The developer must adhere to the CS-351 coding standards.

### 5-2 Format of the SUBMISSION ARCHIVE

The SUBMISSION ARCHIVE must be a **`jar`** file containing the following top-level directories:

**`src/`** Root of all source code (package) directories.

**`bin/`** Root of all compiled **`.class`** file directories.

**`doc/`** Contains all USER DOCUMENTATION and all human-readable INTERNAL DOCUMENTATION (such a design documents).

**`api-doc/`** Contains all INTERNAL DOCUMENTATION generated from Javadoc (i.e., documentation of the code itself).

**`images/`** Contains each of the required images in 16-bit color PNG lossless compresses format.

**`data/`** Directory containing any data used by test scripts. However, the submitted archive MUST NOT contain more than 1 Mb of data.

**`gitlogs/`** Copy of log files generated during the development process

The SUBMISSION ARCHIVE MAY, in addition, contain the following:

**`scripts/`** Any non-java scripts or supporting configuration files. Examples include ant scripts or Eclipse wizard **`xml`** configuration files.

The SUBMISSION ARCHIVE MAY contain other top-level directories, at the developer's discretion, but the purpose of all such directories MUST be documented in the USER DOCUMENTATION (e.g., in the **`README.TXT`** file.)

The SUBMISSION ARCHIVE MUST NOT contain any of the following:

☠ The Eclipse **`.classpath`** or **`.project`** files.

*Hint*: To configure Eclipse to access a **`jar`** file from a project, select the project name in the package explorer, pick "**`Properties`**" from the context menu, go to **`Java Build Path`** → **`Libraries`** → **`Add External Jars`**, and browse to the location where you have stored the file.

For rollout, the SUBMISSION ARCHIVE must be an executable archive such that the command:

```
java -jar TriangleGenome_lastName1_lastName2_lastName3.jar
```

will run the developer’s `TriangleGenome` program.

## 5-3 The Genome

The algorithm of the great **`TriangleGenome`** project begins quite humbly by generating 200 triangles each with randomly placed vertices, a random color and a random transparency. Since we will be using a genetic algorithm, each of the ten properties of each of the 200 triangles shall be considered a *gene*. In a genetic algorithm, the full set of genes that comprise a valid solution is called the solution's *chromosome*, *DNA* or the *genome*. In the **`TriangleGenome`** specifications, a valid solution will be called a genome.

Note: "valid" does not imply optimal nor even near optimal. In the **`TriangleGenome`** specifications, a valid solution is any sequence of 2000 integers (200 triangles $\times$10 values per triangle) where the graphical component that corresponds to each integer is within the valid range for that component. For example, triangle vertices MUST be within the image bounds. The red, blue, green and alpha values MUST be between 0 and 255.

Note: Even though the project specifically specifies that there will be 200 triangles per genome, the developer must never hard-code this value anywhere other than in a single static final field such as:

```
public static final int TRIANGLE_COUNT = 200.
```

## 5-4 Fitness

In natural systems, a genome's fitness is dependent on many factors, is difficult to measure and can change radically when the environment changes. In the **`TriangleGenome`** system, a genome's fitness is a measure of how closely it matches a specified target image. Even in this very simple system, the meaning of "closely matches" could be defined in many different ways. For example, the human eye is more sensitive to differences of intensity in green than in red. Therefore, errors in the green color channel should perhaps be weighted more heavily. Even more complex is the fact that humans are more sensitive to visual errors on, for example, a person's face than errors of the same magnitude in a person's hair, clothing or in the background.

In the **`TriangleGenome`** project, each team must precisely define a ***fitness*** function that mets the following criteria:

1) The fitness function returns zero for a pair of identical images.
2) The fitness function returns increasingly large numbers for images that would generally be agreed are less alike.
3) The fitness function may be undefined for a target / phenotype pair of images of differing dimensions.
4) The fitness function is normalized. That is, if each image in a target / phenotype image pair is resized to half size (using a bicubic algorithm such as is in Photoshop), then both the original pair and the resized pair should have approximately the same fitness.

Generally, calculation of the fitness function dominates execution time of genetic algorithms. In this, **`TriangleGenome`** will be no exception. Thus, many optimization and design decisions should be centered on the fitness function. For example, the developer should avoid unnecessary evaluations of the fitness function by one or more of the following strategies:

1) Eliminating (or at least minimizing) cases where the fitness of two identical genomes is independently evaluated.
2) Recognizing properties or patterns common among unfit genomes so that either they are never created or, once created, are killed off before fitness evaluation.
3) Using lazy evaluation of fitness. That is, designing an algorithm that is able to move toward a solution while maintaining imperfect knowledge of the fitness of many of its genomes.
4) Using difference evaluation. For example, since the required difference function is the summation of differences in all pixels, if a parent and child genome differ in only a relatively small region of the image, then the child's fitness can be exactly determined by adding the difference between child and parent *in that region* to the parent's fitness.
5) Optimizing fitness evaluation. For example, in the fitness function defined above, interchanging the three summations (x, y, and color) has no effect on correctness; however, due to cash coherency, may have a significant effect on execution time.
6) If an image is rendered to calculate its fitness, this must be done in an off-screen buffered image. Even were the program run in a single thread, off-screen rendering is faster than on-screen. In the case of multiple threads this difference will be even more significant.

*Extra Credit*: In Java, the most commonly used off-screen buffer is the **`java.awt.image.BufferedImage`**. There is, however, a higher preforming alternative: the **`java.awt.image.VolatileImage`**. This is a relatively new object in the Java API and through Java 7, remains under active development. It differs from other Image variants in that, if possible, **`VolatileImage`** is stored in Video RAM (local to the graphics card). This allows for *much* faster drawing-to and copying-from operations. There is a price to pay for this advantage: it means that the program's data is not fully protected by the operating system and its contents are subject to changes caused by other applications (such as a screen saver starting up or a window on a different application being displayed or resized). The API for **`VolatileImage`** provides ways to handle this. Still, using this API does require a bit of extra care. Note: if an application that uses **`VolatileImage`** is run on a machine using "integrated graphics" (i.e. has no VRAM), the program will run fine with the **`VolatileImage`** behaving just as would a **`BufferedImage`** with no hardware acceleration.

*Extra Credit*: Some graphics cards offer hardware accelerated image subtraction and difference calculation (exactly the operation required by the **`TriangleGenome`** fitness function). The developer looking for a challenge and who wants the genetic algorithm to run like lightning, might consider JOCL (http://jogamp.org/jocl/www/).

## 5-5 Hill Climbing

In computer science, hill climbing is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

Hill climbing is good for finding a *local optimum* (a solution that cannot be improved by considering a neighboring configuration). In a large and complex terrain of search space, hill climbing is not good at finding the *global optimum* (the best possible solution).

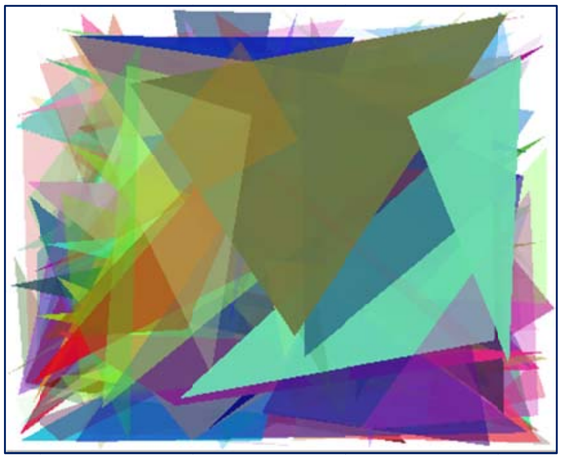
Hill climbing MUST be used in the **`TriangleGenome`** project. In particular, each computation thread MUST start by generating a random genome. Then, it MUST make a "random" change to that genome. If the mutated child genome has a lower (better) fitness value than the parent genome, then the parent is replaced by its child.

The challenge for the DEVELOPER is to find a good definition of "random".

For example, if a particular change is found to make an improvement, then it SHOULD be more likely to make a similar change in the future. Contrariwise, changes that prove detrimental SHOULD become less likely. There are many viable (and many more *possible*) heuristics for this. As it is non-trivial to quantitatively (and in many cases even qualitatively) evaluate these heuristics before implementing them, the DEVELOPER SHOULD strive for a design that is flexible.

## 5-6 Random Starting Triangles

**`TriangleGenome`** MUST seed each worker thread's hill climb algorithm with a sequence of random triangles. However, as discussed earlier, "random" does not mean without thought or guidance. For example:



This 450×363 image shows a sequence of 200 triangles where the *x*-coordinate of each vertex is a random number, uniformly distributed between 0 and 449.

Notice that the triangles are piled up in the middle.

This is because an edge pixel only gets color when a vertex is on that edge. By contrast, an internal pixel gets color whenever vertices surround it.

The developer should implement a method for generating random genomes that does a more uniform job of covering the image than the naive implementation shown above.

The developer may prevent seeding the genome with degenerate triangles (triangles in which the vertices are co-linear or nearly so).

During mutation, a triangle's alpha gene (the transparency) should be able to explore the full range of valid values. However, the developer may decide to limit the range of random alpha values in the *starting* genomes to $n \le alpha \le 255$, $0 \le alpha \le n$, or $n \le alpha \le m$. Where $n$ and $m$ are valid alpha values.

The developer may seed the starting triangles with color samples from the original image. However, this must be done with care: a genetic algorithm depends on a starting population that includes sufficient diversity so that it is possible for a solution to be found by cross-over and rearrangement of the original genetic genes. Over seeding the starting population can over constrain the algorithm.

Note: The background color of the genome image MUST be white.

## 5-7 Genetic Algorithms and Crossover

In hill climbing, mutations are made to a single parent.

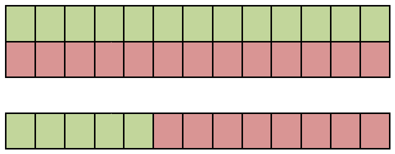
By contrast, a genetic algorithm requires at least two parents and some heuristic for generating a child genome by mixing genes form those parents. This is called *crossover*.

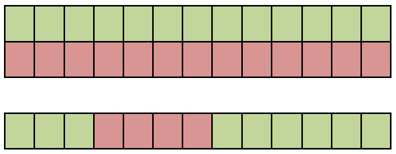
Some common crossover heuristics include:

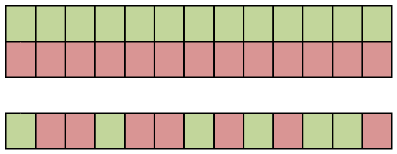
Single-point crossover: The child's genome is created by starting at one end of one parent's genome and copying genes, one at a time, from the parent to the child. Then, at a single random location, switch parents and from that point on through the end of the genome, copy genes from the second parent to the child.

Double-point crossover: The same as single-point crossover, except after switching to the second parent, pick another random point and switch back to copying from the first parent.

Uniform crossover: At each location along the child's genome, randomly select one or the other parent. Copy, to the child, the gene from the selected parent at the corresponding location.


Single-Point Crossover Example


Double-Point Crossover Example


Uniform Crossover Example

There are many variations of crossover. The effectiveness of these variations is highly dependent on both the search space and fitness function.

**`TriangleGenome`** must implement at least two different crossover heuristics and quantitatively compare the results of using each.

The developer may want to consider the merits of allowing cross-over to start at any gene in the sequence verses only on triangle boundaries.

The developer may want to consider the possibility of allowing cross-over to start at any ***bit*** along the genome. For example, the red value of triangle number 13 uses 8 bits to express the number 0-255. Bit level crossover would mean that while coping the 3rd least significant bit of this gene, the parent could switch and the 4th least significant bit of this same gene location from the second parent would be copied.

In a genetic algorithm, crossover fails when it produces a child that has the same gene sequence as one of its parents. As the algorithm converges toward a solution, the genetic diversity in the population often decreases and the likelihood of such failed crossovers increases. Consider, for example, a pair of parents with only 8 of the 2000 genes in each parent having different values and that all 8 of those different valued genes are in the first 200 genes in the sequence. A child is produced by crossover where the first crossover does not start until after gene 200 ***will*** be identical to one of its parents. For this reason, the developer may want to consider implementing not a random crossover gene index, but a random number of differences to be counted before crossover. Hint: the method `int getHammingDistance(Genome dna2)` of the provided **`Genome`** class may be used to help determine the range of differences to count before crossing over with a given pair of parents.

## 5-8 Genetic Algorithms and Selection

Hill climbing uses a population of one.

Genetic algorithms require a population of many. It is generally true, that in order for a genetic algorithm to be effective, the population must be both large and diverse enough so as to contain, sprinkled throughout the population, all genes required for a genome to be somewhere on the slope of the optimal solution in the search space.

The developer should consider ways to maintain diversity in the population.

From this large and diverse population **`TriangleGenome`** MUST employ some method of *selection* that brings two members of the population together to produce a child by crossover. Again, there are almost as many methods of selection as there are genetic algorithms. The general principles that inspire most GA selection methods are:

1) More fit members of the population need to have a higher probability of being selected than less fit members.
2) Every member of the population must have a significant probability of being selected.

## 5-9 `TriangleGenome`: Fitting the Pieces Together

When executed, **`TriangleGenome`** MUST load a target image and instantiate a set of worker threads, *tribes*, as specified in the GUI. Valid values for this field are 1 through 1000.

Each tribe must run in its own thread and be separate from the GUI thread.

Each tribe must be initialized with a random genome.

If the number of tribes equals one, then **`TriangleGenome`** MUST execute with a single tribe and each generation must be produced only by hill climbing (this is the basis of Milestone 2).

If the specified tribes is greater than 1, then **`TriangleGenome`** MUST use both hill climbing and crossover: Each tribe MUST independently run a hill climbing algorithm. The developer must define a trigger that signals all tribes to synchronize parts of their data and commence crossover. Then each tribe must develop a population and continue generations of crossover until that tribe reaches a second trigger. At that point, the tribe MUST switch back to hill climbing. This switching between the local and global search strategies MUST continue until an optimal solution is reached or stopped by the user.

Different developers may choose different triggers such as:

1) When a tribe finishes a fixed number of generations.
2) When a tribe reaches a fixed fitness level.
3) When the average fitness improvement per generation falls below some threshold (the "current method is not working so do something else" strategy).

When one tribe transitions to crossover, all tribes MUST do so.

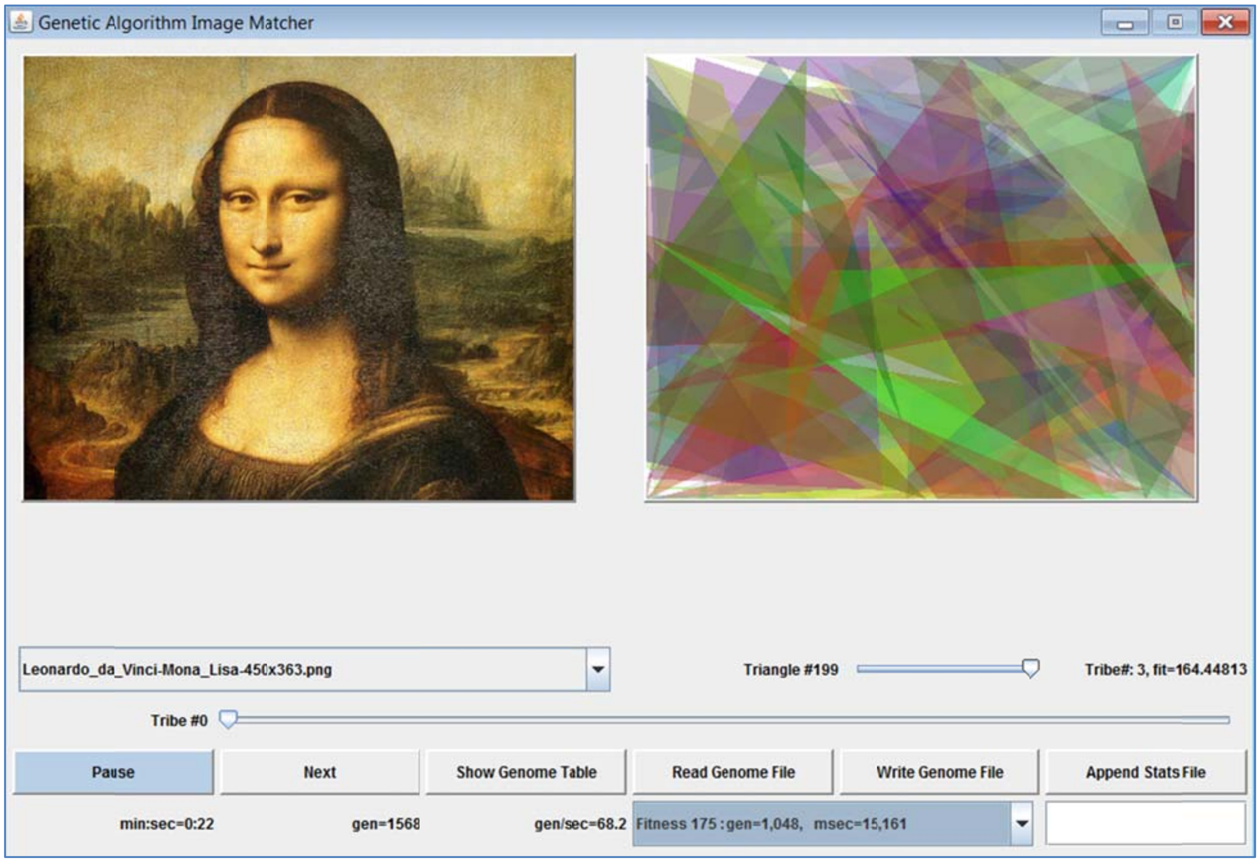
Before crossover begins in a particular tribe, that tribe MUST have a copy of the current best genome from every other tribe's population. Additionally, tribes MAY exchange other members of their populations and/or other data such as data from any adaptive strategies they might be using.

When one tribe transitions from crossover back to hill climbing, the other tribes MAY also transition.

If the number of tribes is greater than or equal to the number of processor cores, then throughout hill climbing, crossover and during transition periods the CPU MUST remain 95+% busy with only short dips down to 85+%.

## 5-10 The Graphical User Interface (GUI)

The `TriangleGenome` GUI MUST include specific components with specific functionality. The layout of the GUI is up to the DEVELOPER. The GUI MUST run in a separate thread from all heavy computation. While the search algorithm is running, any dynamic GUI components MUST be updated approximately once per second.



Example GUI Layout

The GUI MUST include the following components:

Target Image Area: MUST display the image currently selected in the Target Image JComboBox. The target image area MUST be able to display an image up to 512x512 pixels.

Genome Image Area: When the search algorithm is paused, the Genome Image Area MUST display triangles 0 through *n* (where *n* is controlled by the Triangle Slider component and may be any integer through the number of triangles in the genome, 200). When the search algorithm is running, the Genome Image Area MUST display *all triangles* (regardless of the setting of the Triangle Slider) of the best genome existing throughout the current population of all tribes. When the search algorithm is paused, this panel must display the currently selected genome of the currently selected tribe. The Genome Image Area MUST be the same size as the currently selected target image.

Target Image JComboBox: MUST contain each **`String`** listed below. The order of the list is not defined. Each of the items in the list must be an image file in **`images/`** directory. All of these images, except *developer defined*, are supplied on the class website or can be resized from the supplied image. The *developer defined* image should be chosen as being particularly interesting for some algorithmic reason. For example, an image that the developer's implementation of **`TriangleGenome`** preforms particularly well, particularly poorly, produces some interesting artifact, ...). The default selection is not defined. This component MUST be enabled if and only if, the Pause JToggleButton is selected.

"Leonardo_da_Vinci-Mona_Lisa-450x363.png"
"Hokusai-Great_Wave_Off_Kanagawa-200x137.png"
"Hokusai-Great_Wave_Off_Kanagawa-450x309.png"
"Claude_Monet-Poppy_Fields-450x338.png"
*DEVELOPER defined*

Triangle Slider: A **`JSlider`** with values 0 through the number of triangles in the genome. There MUST be some label or other indication that 1) identifies what the slider is and 2) the slider's current selected value. This component MUST be enabled if and only if, the Pause JToggleButton is selected. The default value of this slider MUST be all the triangles in the genome. Whenever the Pause JToggleButton is toggled, this slider MUST be set to its default value.

Genome Info: The GUI MUST display the tribe number and normalized fitness value of the genome currently displayed. The normalized fitness level MUST be displayed with five decimal places.

Total Tribe Count: The GUI MUST have a way for the user to specify the total number of tribes (1 through 1000).

Tribe Selector: The GUI MUST have some way for the user to select which tribe’s data is currently displayed in the GUI.

Genome Selector: The GUI MUST have some way of showing the user 1) how many genomes are currently in the selected tribe’s population and 2) allowing the user to choose which of those genomes to display in the Genome Image Area. The Genomes in this selector must be sorted by fitness. Note 1: If the developer is using lazy fitness evaluation, then it may be that some or all of the finesses in the population are not precisely known and, therefore, some of the genomes may be in the wrong order. This is okay. Note 2: The developer may choose to have all tribes always have the same population throughout the search or the sizes of these populations may vary.

Pause JToggleButton: When selected, all search threads MUST finish their current generation and then sleep. When unselected, all search threads must awaken.

Next Generation Button: When clicked, each search thread MUST awaken, preform exactly one generation and return to sleep. This component MUST be enabled if and only if, the Pause JToggleButton is selected.

Show Genome Table Button: When clicked, the GUI MUST instantiate a window that displays the full genotype of the genome currently displayed in the Genome Image Area. This component MUST be enabled if and only if, the Pause JToggleButton is selected.

Write Genome File Button: When clicked, this must save a copy of the currently selected genome to a file in human readable XML format. DO NOT use object serialization! Serialized objects are not human readable and (after making even small changes to a program) are generally not even Java readable! This component MUST be enabled if and only if, the Pause JToggleButton is selected.

Read Genome File Button: When clicked, this must display a file dialog and if the user selects a saved genome, then the genome must be loaded, inserted someplace into the population of the currently selected tribe and displayed in the Genome Image Area. When this is done, all relevant fields and GUI components must be updated to reflect the current state of the program.

Note: when a genome file is loaded, all current and accumulated statistics MAY be considered meaningless.

If this action fails or the read genome does not match the size of the selected target image, then an appropriate error message MUST be displayed.

This component MUST be enabled if and only if, the Pause JToggleButton is selected.

Current Statistics: The GUI must display and maintain updated values of the following statistics:

1. Total Elapsed Time: MUST be the sum of elapsed wall clock time during which the Pause JToggleButton has been unselected. The Total Elapsed Time MUST be set to zero when a target image is selected. This must be displayed in the format **`m:ss`** where **`m`** is an integer number of minutes (0-9,999) and **`ss`** is an integer number of seconds (0-59).
2. Total Generations: MUST be the sum of all generations (both hill climb and crossover) of all tribes. Total Generations MUST be set to zero when a target image is selected.
3. Hill Climb Generations: MUST be the sum of all hill climb generations on all tribes.
4. Crossover Generations: MUST be the sum of all crossover generations on all tribes.
5. Generations per Second: MUST be equal to (Total Generations) / (Total Elapsed Time) accurate and displayed to ***one*** decimal place.
6. Tribe and Total Population Delta Fitness per Minute: Must display the change in fitness of the most fit genome during the most recent minute (wall clock) in both the currently selected tribe and in the population as a whole. Note: this is undefined from the start of a search until the first minute has passed.
7. Tribe and Total Population Diversity: Must display some measure of the currently selected tribe and of the total population diversity at the current time within the search.

Accumulated Statistics: The GUI MUST have some way to maintain and display accumulated statistics of the above defined *Current Statistics*.

## 6 Non-Project Specific Requirements

1) **`TriangleGenome`** MUST NOT crash, core dump, dump a stack trace, or throw an exception on any input.
2) The DESIGNER MUST adhere to good coding style. This includes consistent use of capitalization and naming for classes, methods and fields. Good coding style includes consistent indenting and other formatting, removal of dead code, not repeating large blocks of near identical code, breaking logical code blocks into methods, and providing adequate comments. To the extent possible, the code style MUST be "self-documenting". The DESIGNER's style need not be consistent the style used in classes provided by the instructor or other authors.
3) In the case of a RECOVERABLE ERROR, the program MUST issue a warning statement and continue processing. The program MAY choose to issue the warning statement to standard error, to a log file, or to a user interface element. If the warning is issued to a log file, the log file name and location MUST be a user-specifiable parameter to the program (either by command-line command or via a configuration menu).
4) In the case of an UNRECOVERABLE ERROR, a program MUST issue an error statement and terminate with a non-zero error condition. The program MAY use different exit codes to indicate different error conditions, but such codes MUST be documented in the user manual. The error message MUST be logged to the same destination that warning messages (from RECOVERABLE ERRORS) are. In the case of any ERROR, a program MUST NOT delete, corrupt, or damage existing data files or any other “stateful” files employed by the program suite.
5) The programs MAY provide additional output for debugging purposes, but such output must be disabled by default. Any program MAY provide a command-line switch or a user interface configuration utility to enable debugging support when desired.
6) All user documentation MUST be grammatically correct and include correct spelling and usage. (You will be graded, in part, on the quality of your writing.)
7) The designer MUST document any areas in which her or his software suite does not meet this specification. WARNING! The grade penalty will be higher if the instructors discover an undocumented program shortcoming or bug than if it is documented up front.

## 7 Deliverables and Milestones:

See Slides