

# A Java Program to Create Simulated Microarray Images

Bill Martin <eco\_bill@yahoo.com>

Robert M. Horton, Ph.D. <rmhorton@attotron.com>

*Attotron Biosensor Corporation*

## Abstract

*We have developed a microarray image generation program to create customized sets of images representing "virtual microarray experiments" to teach microarray technology in undergraduate biology courses.*

## 1. Introduction

Microarray images usually represent large amounts of data containing a wide assortment of imperfections. These can be daunting to students seeking an introduction to the technology. Simulated "toy" images can be customized for particular teaching points. For example, a set of idealized images makes it easy to move from images to measurements to clustering and visualization within a single laboratory period. In more advanced exercises, variable spot placement can be used to develop gridding skills, faint spots to teach about signal-to-noise ratios, variable feature shape to teach segmentation methods, etc. Anomalies can be simulated in combinations to produce arbitrary degrees of realism.

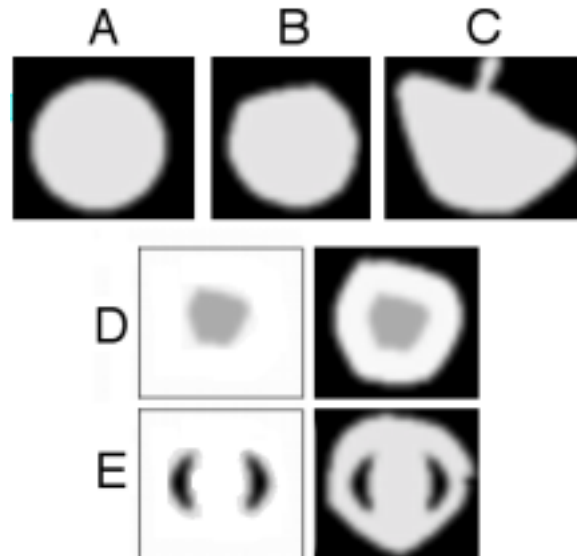
Instructors can design arrays by specifying reporters and their spatial arrangement, or choose from a catalog of existing designs. Relative spot intensities can be either specified by an instructor, or extracted from actual experimental data repositories such as the Gene Expression Omnibus[1]. If experimentally observed data are used, students can compare their results from analyzing simulated images to the original papers in which the experiments were described.

A pair of grayscale TIFF images represent the red and green channels of a two-color experiment. They can be measured with available software such as ScanAlyze[2] or MagicTool[3]. Batches of "experiments" using one array design with a series of virtual RNA samples can be delivered in a ZIP file.

## 2. Object Model

A **Microarray** object coordinates the construction of

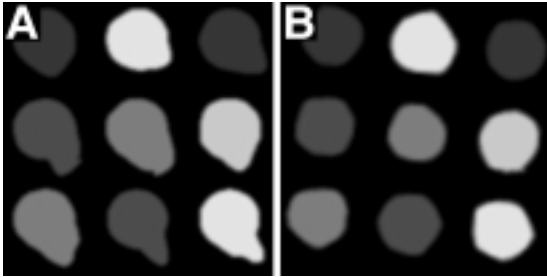
a rendering chain of events, managing image housekeeping details. It associates an array of features with a background image and a **FeaturePainter**. The **Scanner** encapsulates details about how the final rendering is to be done (such as gain and pixel size) and saves the results to a pair of TIFF images and/or a JPEG file. A **FeatureData** object encapsulates the description of a feature in the microarray hybridization. It includes a location (in micron coordinates) and nominal channel intensities (representing the degree of hybridization).



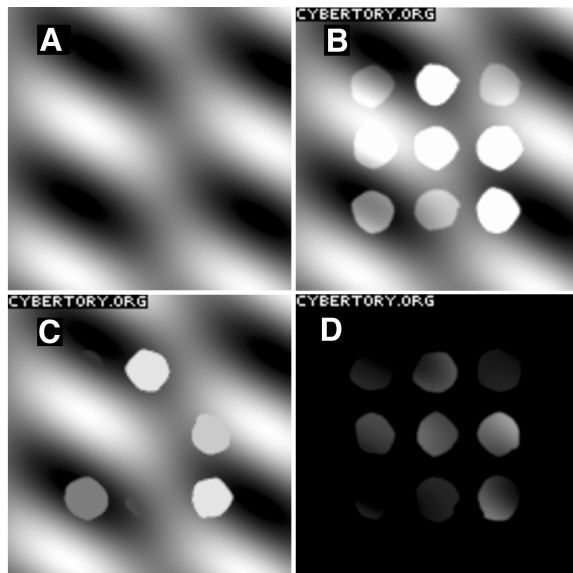
**Figure 1: Variability in Feature Shape and Texture.** *Shapes.* **A.** Perfect circle uniformly filled with the proper grayscale level. **B.** Random polygons with rounded vertices create variable shapes. **C.** Wildly erratic feature shapes might be useful to demonstrate or test segmentation algorithms. *Textures.* A texture is multiplied (pixel-wise) by the uniform intensity. In this case, the average intensities over the spot are (approximately) preserved so that measured intensities are similar to the input values. On the left are the FeatureTexture images, and on the right the simulated features. Note that textures are abstract and are shown here for illustration only. **D.** halo or donut effect. **E.** "pig nose" effect observed in some real-life images.

The default implementation of a FeaturePainter

uses a **FeatureShapeFactory** and a **FeatureFiller**. A **CircleFeatureFactory** makes circular features, and when shape variability is desired it delegates to a **PolygonFactory** which generates random polygons to approximate imperfect spots or "blobs". Blob size and irregularity are controlled parametrically, and corners can be rounded by connecting vertices with cubic curves. A feature may be filled with a uniform intensity, or with a **FeatureTexture** multiplied (pixel-wise) by an intensity.



**Figure 2. Grid Variability.** **A.** The **PolygonFactory** supports a configurable bias in the radius near a given angle for directional "smearing." **B.** Grid rotated slightly around the center of the microarray.



**Figure 3. Image Background Compositing.** **A.** A "blotch" image is produced with an **ImageFunction**, calculating pixel values by position, such as:  $\text{intensity} = 0.5 + 0.25 * (\sin(-0.08 x + 0.08 y) + \sin(0.0084 x + -0.0880 y))$  **B.** features and background composited with the "add" operator. **C.** composited with the "max" operator, taking the brighter pixel of the 2 images. **D.** composited with the "multiply" operator.

### 3. Java Advanced Imaging (JAI) API

The image generator uses the Java Advanced Image (JAI) API, a very powerful and flexible image creation and analysis tool. A wide variety of imaging operations are available, and the framework allows for easily adding custom operations. Image data (pixels) can be stored and manipulated as integers of any size from byte to long, or floating point numbers (regular or double precision).

Floating point image representation allows both high precision and useful abstractions, including negative or supersaturated pixel intensities and operations like multiplying two images together. In such operations, an abstract "image" might represent data for manipulation but have no meaning in terms of display.

### 4. Research Applications

In addition to educational applications, simulated images may be useful for testing and evaluating image analysis tools. Scientific image analysis relies on conceptual models that encapsulate assumptions about factors comprising the image[4]. For example, an analytical model might assume that feature intensity is added to background signal, or that "noise" is normally distributed. Because simulated images can be reverse-engineered to exactly instantiate such models, they may prove useful for "black box" evaluation of analytical software products, or for testing of new software.

### 5. Availability and Acknowledgements

The microarray image generator is part of the Cybertory Virtual Molecular Biology Laboratory project ([www.cybertory.com](http://www.cybertory.com)), and is released under the GNU Public License. It is written in Java using the Java Advanced Imaging (JAI) API. This work is funded by NIH SBIR grant 2R44RR013645-02A2 to Attotron Corporation.

### 6. References

1. Gene Expression Omnibus [www.ncbi.nlm.nih.gov/geo](http://www.ncbi.nlm.nih.gov/geo)
2. ScanAlyze <http://rana.lbl.gov/EisenSoftware.htm>
3. MagicTool. [www.bio.davidson.edu/Biology/MAGIC/MAGIC.html](http://www.bio.davidson.edu/Biology/MAGIC/MAGIC.html)
4. Kamberova, G. and Shah, S. (eds) DNA Array Image Analysis Nuts and Bolts. DNA Press, 2002.