Journal of Applied Mathematics and Computational Mechanics 2013, 12(2), 5-13

# GENERATING PLACATED RANDOM SHAPES FOR AN AREA ESTIMATION STUDY

*Abdullah Almowanes*[1], *Tamar Kakiashvili, MD*[2], *Waldemar W. Koczkodaj*[1]

[1] *Computer Science, Laurentian University, Sudbury Ontario, Canada*
[2] *Baycrest, Brain Research, Toronto, Canada*
*wkoczkodaj@cs.laurentian.ca*

**Abstract.** Random but visually nice shapes are often needed for cognitive experiments and processes. This study describes a heuristic for generating random but nice shapes. We call them placated shapes. These shapes are produced by applying the Gaussian blur to randomly generated polygons. Subsequently, the threshold is set to transform pixels to black and white from different shades of gray. This transformation produces placated shapes for easier estimation of areas. Randomly generated placated shapes are used for testing the accuracy of cognitive processes by pairwise comparisons. They can also be used in many other areas such as computer games or software testing. Such shapes could also be used for camouflaging heavy army equipment.

***Keywords:*** *image processing, Gaussian blur, random polygon, computational geometry, cognitive process, software testing*

## Introduction

Our study demonstrates an algorithm for generating placated random shapes. No one really knows what a nice shape is. However, we can recognize nice shapes once we see them. The observer can judge how successful we were in our attempt. Generating totally random shapes is easy: generate random coordinates, place one pixel, and keep adding other random pixels to it. We have not done it here since we need random shapes that are not only nice but also not too difficult to estimate their areas for a specific experiment (pairwise comparisons). Amongst many others, random shapes are needed for testing and computer games. A 1D case (randomly generated bars) for testing the accuracy of pairwise comparisons was published in [1] and [2] as the first statistically correct study in the world. The random bar length estimation error went down from approximately 15% (by direct method) to approximately 5% by pairwise comparisons method.

Evidently, it was not easy to find a solution to a 2D case since random but nice shapes needed to be generated. A diabolical experiment was designed and published in [3] where random but equal in area shapes were used. However, the shapes, although random, were a product of a human hand (as described in [4])

where “a combination of rough pencil and brush strokes were used for Gaussian blurring” was proposed by the second author as his major contribution to [4]. Respondents were tricked into comparing random but equal shapes (according to the area) without knowing that all shapes had identical area. This cognitive experiment was designed by Koczkodaj and reported in [3]. The results of the cognitive experiment were rather astonishing. Only a few respondents guessed the area equality. The estimation of one area shape to be up to 10 times bigger than area of another shape was not infrequent.

## 1. Description of a heuristic for placated random shape generation

### 1.1. Rationale

Our shape generation begins with generating a random polygon. It can be degenerated (e.g., with holes). Several heuristics were presented for the random generation of polygons and implemented as the RPG (Random Polygon Generator) software package [5]. Although it is possible to generate nice random shapes, the RPG heuristic algorithms are considerably complex to implement. They are designed to generate polygons with specific features from arbitrary sets of prescribed vertices. Our approach is more efficient. There is no need to compute polygons because we blur them and cut at a randomly chosen threshold to obtain smoothness. In particular, there is no need to enforce topological shape constraints. So, we make no effort to ensure that the polygons are simply connected (in the sense of topology), since this property may be lost after blurring. In our approach, connectedness follows from generating a closed polygonal curve from the seed points, and it is maintained by blurring and thresholding with a sufficiently permissive cut-off. Obviously, the results depend on the number of points N and how we generate random coordinates. The method used to generate a curve from the random coordinates, the blur radius, and the threshold value also influence the look of shapes.

In principle, any interpolatory or smoothing curve generation method (B-splines, Bézier, Hermite, etc.) could be used to generate the curve from the random points. We use the simplest linear spline interpolation, a.k.a. “join the dots with straight lines”. The curve should not be too thin, otherwise the final thresholding may introduce aliasing artifacts when the blur is applied. Blurring gives a smoother intensity surface, which leads to a smoother shape after post-thresholding. The number of randomly generated points N should be large enough so that the polygonal curve is non-trivial. We recommend using at least ten points unless N is itself a random variable. By letting N take lower values, very simple shapes would be generated. On the other hand, N should not be so large that random variations in the random coordinates are averaged out by the blurring process, resulting in a featureless blob.

A Java random number function has been used for point coordinates by rescaling the random number. It has been done in such a way that the points fit

in the assumed canvas which is in our case 250 by 250 pixels. In our prototype, we have used a cut-off algorithm which is not optimized. It takes (on average) two minutes to generate one picture, even for a small canvas. Blurring adds the "mild and soft" look to an initial rough shape. Attractive for its simplicity, this approach has great potential. As of now, the points are uniformly distributed in a square and the collection of generated shapes is not rotation invariant for any point of the figure. As a consequence, the generated shapes have angles which are not uniformly distributed, which is easy to fix. Instead of drawing the points from a uniform distribution, we draw them from a rotation invariant distribution.

### 1.2. The random shape heuristic algorithm

It is a natural temptation to draw a swirling line or a curve when we attempt to generate a "random but nice" shape. However, designing an algorithm or a heuristic for such a "random line" is not easy. Certainly, we can go left, right, up, or down randomly and we can also choose a random number of steps but such a line will be very rigid and may cross itself many times. Table 1 shows a heuristic algorithm which we have implemented.

Table 1

**Placated random shape generation heuristic algorithm**

| Generate random polygon |
|---|
| Apply Gaussian blur with a randomly chosen parameter |
| Select random threshold for turning pixels to black (with values below the random threshold) and white (above the random threshold) |

### 1.3. Gaussian blur example

The Gaussian blur is obtained by applying the following Gaussian function:

$$g(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2e^2}} \tag{1}$$

where $x$ is the distance from the origin in the horizontal axis, $y$ is the distance from the origin in the vertical axis, and $\sigma$ is the standard deviation of the Gaussian distribution. By "smearing" an image with a Gaussian blur, we make it nicer. In essence, each pixel is mapped into a weighted average of that pixel's neighborhood. The highest Gaussian value (weight) is given to the original pixel, while the neighboring pixels receive smaller weights, since their distance to the original pixel increases. Here is an illustration example of using a program written in Java that applies a Gaussian blur to a randomly created shape and a 15 x 15 Gaussian filter. The kernel used for blurring Figure 1 is (2).

$$\begin{bmatrix} 0.0009 \\ 0.0016 \\ 0.0027 \\ 0.0045 \\ 0.0071 \\ 0.0108 \\ 0.0158 \\ 0.0222 \\ 0.0300 \\ 0.0389 \\ 0.0485 \\ 0.0581 \\ 0.0668 \\ 0.0739 \end{bmatrix} * \begin{bmatrix} 0.0009 \\ 0.0016 \\ 0.0027 \\ 0.0045 \\ 0.0071 \\ 0.0108 \\ 0.0158 \\ 0.0222 \\ 0.0300 \\ 0.0389 \\ 0.0485 \\ 0.0581 \\ 0.0668 \\ 0.0739 \end{bmatrix}^T \tag{2}$$

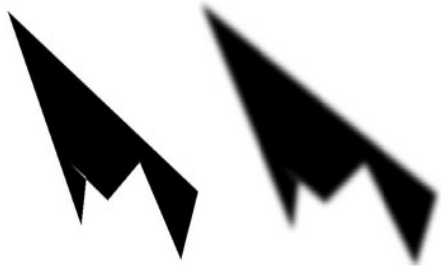

Fig. 1. Before and after blurring

Figure 1 shows a randomly generated shape before and after applying a Gaussian blur with a 15 radius. The smoothness is apparent in this figure.

### 1.4. Dealing with holes

Shapes that are randomly generated with the growing number of points N are expected to have holes in them. To get a nice-looking shape without holes, we can adjust the parameters to get the most desired looking shape. By the parameters, we understand the Gaussian blur radius and the threshold value. In the first example, Figure 2a, the shape is randomly generated and constructed by 7 points. As we can see, the shape does not look excessively nice. They look like three different, but not completely, connected shapes. Getting a nice smooth-looking shape requires some image processing. First, we will apply the Gaussian blur of different radius values and then we will apply a threshold with the appropriate pixel value.

Applying a Gaussian blur with a radius of 20 produces Figure 2b. The shape is more connected now but there is a large hole in the center. By applying a threshold of 100, we get the shapes shown in Figure 2c. There are three separate shapes which are not desired. The larger threshold brings us closer to our goal. For the 200 threshold, the hole is still there (Fig. 2d) while our goal is a randomly generated nice looking shape with no holes. However, the hole is large, hence the threshold value should still be larger to eliminate the hole. The threshold 254 is still insufficient in removing the hole. Finally, the threshold value 255 is good enough to remove the hole. Unfortunately, the shape size has increased and some parts of it are no longer visible hence this violates the "niceness" of the shape.

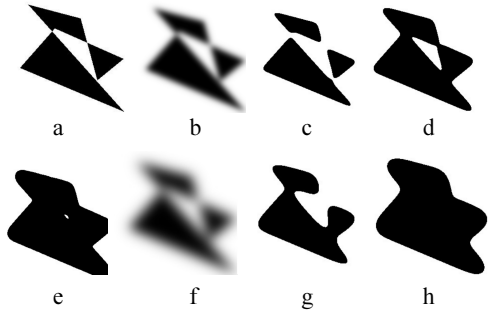


Fig. 2. Example using 7 points shape

Figure 2e shows the hole for the 254 threshold. Processing the same shape, a larger Gaussian blur radius with an appropriate threshold, can remedy the problem. Figure 2g shows the processed image after applying a Gaussian radius of 40 and a 127 threshold. It is all connected and without holes as desired, but what about the two strange-looking narrow nicks in Figure 2g. To eliminate them, a larger threshold must be used. The magic threshold number for this particular shape that will result in the desired shape, is 218. Using a 40 Gaussian radius and the 218 threshold will produce the placated smooth-looking shape that is randomly generated (Fig. 2h).

Figure 3 demonstrates a different example where the randomly generated shape is sharp and has a large hole in the middle (Fig. 3a). You can see that the area covered in black is smaller than the one it was in the previous original shape (Fig. 2a). The white hole here is smaller. The Gaussian blur with a smaller radius makes it possible to get the desired image. The value can be as low as 10 but the threshold must be 254 for this to work. The blurred image looks like Figure 3b and after applying the threshold is Figure 3c. As you can see, the shape is not very "nice". A larger Gaussian filter will make things better, Figure 3d with a Gaussian radius of 35. A threshold of 240 will result in the following smooth placated looking shape (Fig. 3e).

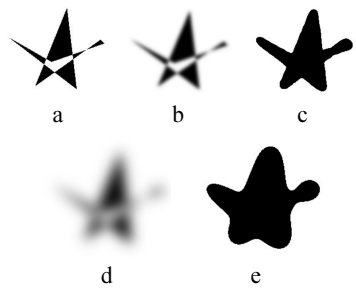


Fig. 3. Example using a star looking random shape

In this third example, there is an 8 points shape with 2 holes (Fig. 4a). By applying a Gaussian blur filter with the value of 25, the result will be a shape in Figure 4b. The threshold 127 generates two shapes (Fig. 4c). This is not acceptable. By increasing the Gaussian blur radius to 45 and setting the threshold value to 150, we get a shape without holes (Fig. 4d). Whenever the threshold increases, the two "arms" of the shape come closer to each other until they meet and construct a new hole. Figure 4e is the shape when the threshold is 157. By increasing the threshold to 206, we get a placated shape with no holes (Fig. 4f).

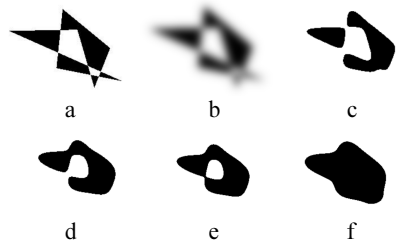


Fig. 4. Example using a 8 points random shape

The next example is more extreme. With too many holes in the randomly generated shape, a higher Gaussian blur radius value is needed to get the nice shape desired. Here, a 75 Gaussian radius is used with a 182 threshold (Fig. 5).

Fig. 5. Shape with more holes

In this last example, a 50 points random shape was generated. The number of holes here is very difficult to count. By applying a Gaussian blur with a 50 radius

and a 127 threshold, the shape shown in Figure 6a is produced. To get the nice shape out of it, a 150 Gaussian blur radius and a 127 threshold are used for the shape in Figure 6b.

a

b

Fig. 6. Shape constructed using 50 points with too many holes

## 2. Java Implementation

Java has been chosen for implementation since its applications are typically compiled to bytecode (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture. It was important considering the results of this project will be used for another project related to pairwise comparisons and data gathering will take place by the Internet and smart phones.

### 2.1. Generate the random shape

First, the random number of points is generated by:

```
int randomNumberOfPoints= min + (int)(Math.random() * ((max - min) + 1));
```

Afterwards, we can generate an array of type Point with the size of the number of point and to generate the x and y coordinates we used the following code:

```
Point[] points = new Point[randomNumberOfPoints];
RandomPoints rp = new RandomPoints(randomNumberOfPoints, imageSize);
```

We took into consideration the image size so we can accommodate all parts of the image after the Gaussian blur is applied. Then, the fillPolygon method was used to draw the shape.

## 2.2. Applying the blur and the cut-off

Applying the blur was done with the help of the ConvolveOp class illustrated in Figure 7. Performing the cut-off was done by checking each pixel and setting the pixel value to 0 (black) for values below the threshold and to 255 (white) (Fig. 8). It will be optimized in time.

```
public static ConvolveOp
gaussianFilter(int   radius, boolean
horizonal)
{
       int size=radius*2+1;
       float data[]=new float[size];
       float sigma=radius/ 3.0f;
                     // radius = 3*sigma
       float woSigmaSquare=2*sigma*sigma;
       float root = (float)
       Math.sqrt(twoSigmaSquare*Math.PI);

       float total=0.0f;

       for(int i=-radius;i<radius;i++){
              float distance=i*i;
              int index=i+radius;
              data[index]=
              (float)Math.exp(-distance/
              twoSigmaSquare)/root;
              total+=data[index];
       }

       for(int i= 0;i<data.length;i++){
              data[i]/=total;
       }

       Kernel kernel=null;
       if(horizonal){
              kernel=new
       Kernel(size,1,data);
       }
       else{
              kernel=new
              kernel(1,size,data);
       }

       return new
       ConvolveOp(kernel,ConvolveOp.EDGE_
       NO_OP,null);
   }
```

Fig. 7. Gaussian blur

```
public    int[][]    ReplacePixel(int[][]
src, int oldValue, int newValue) throws
IOException
   {
      for(int i = 0 ; i < imageHeight;
      i++){
            for(int   j   =   0   ;   j   <
            imageWidth ; j++)
            {
             if (src[i][j] >= oldValue &&
             src[i][j] <=255){
                 src[i][j] = newValue;
             }
return src;
}
```

Fig. 8. The cut-off

## Conclusions

The beauty of the proposed random generation of placated but random shapes is the simplicity of the proposed new image processing method. However, it is worth noticing that the proposed method is simple yet not simplistic. Many former attempts, such as: Bézier parametric curve, Voronoi diagrams, and Turtle graphics, have failed in the past. The attempt will be made to have it as a plug-in for open source image processing systems (such as GIMP) when the software is perfected and extended to other filtering methods and includes the bump function. The results of this study will be used to test the accuracy of pairwise comparisons using 2D random but not equal in area shapes.

### Acknowledgments

The first author would like to acknowledge the endeavours of the sponsor: the Ministry of Higher Education in the Kingdom of Saudi Arabia.

## References

[1] Koczkodaj W.W., Statistically accurate evidence of improved error rate by Pairwise Comparisons, Perceptual and Motor Skills 1996, 82(1), 43-48.

[2] Koczkodaj W.W., Testing the accuracy enhancement of Pairwise Comparisons by a Monte Carlo experiment, Journal of Statistical Planning and Inference 1998, 69(1), 21-32.

[3] Adamic P., Babiy V., Janicki R., Kakiashvili T., Koczkodaj W.W., Tadeusiewicz R., Pairwise Comparisons and visual perceptions of equal area polygons, Perceptual and Motor Skills 2009, 108(1), 37-42.

[4] Koczkodaj W.W., Robidoux N., Tadeusiewicz R., Classifying visual objects with the method of pairwise comparisons, Machine Graphics & Vision 2009, 18, 143-154.

[5] Auer T., Held M., Heuristics for the generation of random polygons, [in:] F. Fiala, E. Kranakis, J.R. Sack, (eds), Proc. 8th Canad. Conf. Computat. Geometry, Carleton University Press, Ottawa, Canada 1996, 38-44.