PROCESSING.JS

Processing.js - Gaussian Blurs with Processing.js

(IE mode)

Gaussian Blur

Hover over the source image to blur the kernel.

Source

Kernel

Blurred

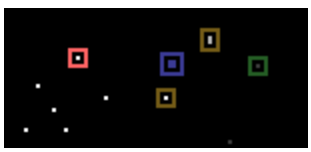
What is a Gaussian Blur?

I am assuming that if you are reading this article, you already know what a Gaussian Blur is. On the chance that you do not... use the example above and you will see it first hand. There are many types of blur out there, but a Gaussian hold special properties due to the mathematics involved. A Gaussian Blur is more 'realistic' looking. It has less artifacts (visual errors); but unfortunately it is usually too slow to use in motion graphics or computer games.

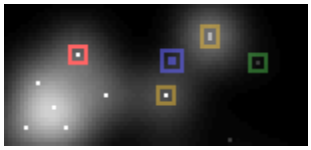
Why bother if it's slow?

Brightness Point Data

Point Data Blurred Twice



Estimate Val of Blue Box?



Use Val of Gaussian
Blur

While it may be too slow to use again and again on large animated images it holds special properties which make it great for still image processing and more contextually, for visual data representation. In a typical situation, it is sufficient to take a range of values which can be represented using a brightness scale. Black being a low value and white being a high value. This technique is most famously used in a satellite generated height map, recording the elevation of terrain; but can be used to represent any data set possessing 3 ranges. This 'normal' map can then be used to accurately estimate random values.

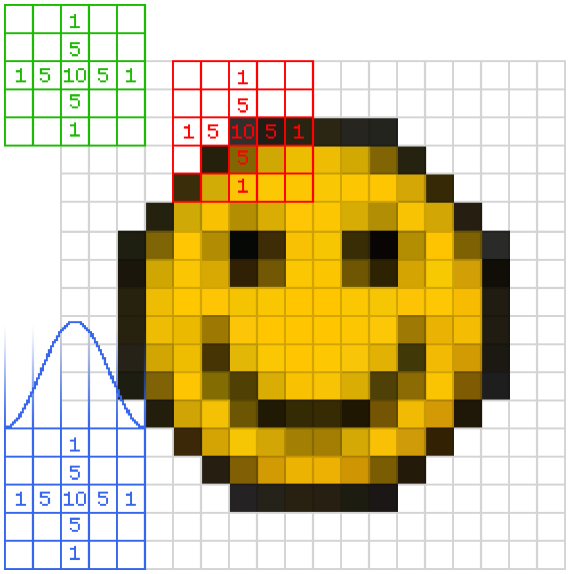
For example: You can see a cluster of data points on the left. The point inside the red box is very bright and the point inside the green box is very dim. If you wanted to estimate the brightness of a point half way between the red and the green boxes (the blue square), would it be accurate to take an average of the two points derived from the distance between them? No.

Why not? The reason this would not be accurate is because there are two brighter dots in the brown/orange squares along the way. How do you add these values into the equation without an insane amount of processing? You average every possible point in the map using a Gaussian curve. Then you can reliably estimate the value, adjusting the spread or 'kernel radius' of the blur to lock down the probability of a correct guess.

And the beautiful thing about blurring brightness... unlike an RGB image it only requires the calculation if one channel. Additionally you can speed up the processing time by using a high-pass, ignoring points with **low** or **zero** value.

Gaussian Blur Theory & Practice

The theory behind a Gaussian blur is fairly simple to understand. While the theory make be very well defined scientifically, the practice of Gaussian Blurring is most definitely an art-form. There are many ways to handle edge cases, such as limiting, clamping, off-page buffering and looping polar co-ordinates. There are also many ways to apply the math: some are faster, some yield greater accuracy. But what are the basic steps you need to take when creating a Gaussian Blur?



1. **Create a Kernel**

   The first step is to create a kernel. A Kernel is a grid through which you filter the color values. The inner-most pixel is called the **"destination pixel"** and it receives colors from the surrounding **"source pixels"**.

2. **Weight The Kernel**

   Next you will need to **weight** your kernel. This is a fancy way of saying... "Let's decided how much color the destination pixel can steal from it's surrounding source pixels". A Gaussian Blur gets it's name because uses a Gaussian or **'bell'** curve. For more information on how I am generating the weights; you can take a look at my previous article Bell Curves.

3. **Separate Blur Axes**

   The primary way to cut the render time of your Gaussian blur is to use **'Separable Passes'**. Rather then apply your entire kernel over every single pixel of your image, you can take a very neat shortcut. Firstly you blur the pixels on the X axis and store the results in an X-buffer; then blur the pixels on the Y axis storing them in your Y buffer.

4. **Process The Kernel**

   Lets say you have a 5-pixel-wide kernel with a Gaussian weight of **[1, 5, 10, 5, 1]** like in the smiley image. When the kernel was calculating the new color value for the middle / **destination** pixel, it would suck some of the color out of the surrounding pixels using the following algorithm:

   destinationColor = destinationColor + (sourceColor * sourceWeight) / gaussianSum

   This equation is calculated for each channel whether it be RGB, HSB, CMYK or even RGBA. One thing to note, is that you can build a **very** fast Gaussian blur algorihm if you only need to blur the luminance channel: which I will cover later. The **Gaussian-Sum** variable is the sum of the numbers in your kernel. Or in our case, it's the sum of a separable axis.

1 + 5 + 10 + 5 + 1 = 22

5. **Handle Edge Cases**

   You will also need to write some code to handle **'edge-cases'**. An edge-case handler is the section of code that deals with how the kernel effects the edges of the image. If you were to move the kernel in the Smiley picture so that the destination pixel was at (0,0) on the grid; you would notice that approximately 3/4ths of the kernel would land outside the image. Depending on how you store your image data, this might return a black color value, a NaN, NULL or even throw an **Array out of bounds** error.

   The most *PhotoShop* looking method I found was to create extra pixels of the Canvas of each image, half the kernel size, and spread or extrude the edge pixels into this zone, so that the blur did not look boxy when the kernel was large. But in the interest of speed I developed an extremely simple approach that could be easily applied to the 1-dimensional array of colors that Processing uses. I will coin it **The Shift Clamp**.

   The way the Shift Clamp works, is to check whether the current source-pixel is out-of-bounds. If it is, the source color is taken directly from the value of the edge color without any extra array space or additional weighting algorithm being used to balance the values. This can unfortunately create some boxy artifacts around the edges of the image in some situations, but I found the trade-off between speed and quality to be more than acceptable in this case.

   As I mentioned previously, there are many, many ways to handle edge cases. You can use polar co-ordinates to **loop** or **wrap** the blur to the adjacent edge. You can apply a harsh limit to the edge, making your blur stop-dead or drop-off as it approaches the edge.

6. **Iterate Kernel Across Image**

   Once you have covered all your bases, you can slide your kernel across the image data array, building your blurred X-pass. Then you slide your kernel down the array, this time taking values from the blurred X-pass, not the original image data!

7. **Update Pixels**

   Finally, you write the Y-pass out to the original image array and update the pixels to your Canvas. That's it! Your Gaussian blur is complete.

A Processing.js Gaussian Blur Demo

I designed a Gaussian Blur demo / code example that you can use in both Processing and Processing.js. This is a 3-channel blur (RGB) which uses Shift-Clamping and includes a Gaussian Curve generator for calculating the Kernel weights on the fly. **Click the Canvas on the left to apply the Gaussian blur:** results typically take **5 - 10 seconds** depending on the processing power of your machine.

If you want to get this working in Processing.js, you will need to update your Processing.js library with the **verifyChannel()** functions I've have parallel native Processing behavior. You will also need to

update the function **updatePixels()** to return the correct alpha range. You can find these in my Processing.js fork on GitHub.

view plaincopy to clipboardprint?

```
1.  // Gaussian Blur Demo - F1LT3R @ Hyper-Metrix.com
2.
3.  // Set the diameter of the Gaussian Blur
4.  int blurDiam = 4;
5.  // Set the width of the Gaussian Blur kernel
6.  int gaussWidth = (blurDiam * 2)+1;
7.  // Create the kernel[array]
8.  float[] kernel = new float[gaussWidth];
9.
10. void setup(){
11.   size(50,50);
12.   background(0);
13.   drawShape();
14. }
15.
16. void draw(){
17. }
18.
19. // Store mouseDown variable when pressing mouse
20. boolean mouseDown = false;
21. void mousePressed(){
22.   mouseDown=true;
23. }
24.
25. // If the mouse is released while in down state...
26. void mouseReleased(){
27.   if (mouseDown == true){
28.     mouseDown=false;
29.     buildKernel();
30.     gaussBlur();
31.   }
32. }
33.
34. // Draw RGB circles
35. void drawShape(){
36.     noStroke();
37.     fill(255, 0, 0);
38.     ellipse(25,15,20,20);
39.     fill(0, 255, 0);
40.     ellipse(35,30,20,20);
41.     fill(50, 50, 255);
42.     ellipse(15,30,20,20);
43. }
44.
45.
46. void buildKernel(){
```

```
47.
48.    // Set the maximum value of the Gaussian curve
49.    float sd = 255;
50.
51.    // Set the width of the Gaussian curve
52.    float range = gaussWidth;
53.
54.    // Set the average value of the Gaussian curve
55.    float mean = (range / sd);
56.
57.    // Set first half of Gaussian curve in kernel
58.    for (int pos = 0; pos < blurDiam+1; pos++){
59.
60.      // Distribute Gaussian curve across kernel[array]
61.      kernel[pos] = sq(sin((((pos+1)*HALF_PI) - mean) / range))*sd;
62.
63.      // Mirror the kernel values
64.      kernel[gaussWidth-1-pos] = kernel[pos];
65.    }
66.  }
67.
68.  // Set Blur variables for source and destination pixels
69.  float sourceR, sourceG, sourceB;
70.  float destR, destG, destB;
71.  int shift, dest, source;
72.
73.  void gaussBlur(){
74.      int timeStartMS = millis();
75.
76.      // Calculate the sum of the Gaussian kernel
77.      float gaussSum = 0;
78.      for (int n=0; n < gaussWidth; n++){
79.        gaussSum += kernel[n];
80.      };
81.
82.      // Copy the current Canvas to the pixels[array]
83.      loadPixels();
84.
85.      // Create an X & Y pass buffer
86.      color[] gaussPassX = new color[pixels.length];
87.      color[] gaussPassY = new color[pixels.length];
88.
89.      // Do Horizontal Pass
90.      for (int y = 0; y < height; y++){
91.      for (int x = 0; x < width; x++){
92.        dest = y*width+x;
93.
94.        // Iterate through kernel
95.        for (int k=0; k < gaussWidth; k++){
96.
97.          // Get pixel-shift (pixel dist between dest and source)
```

```
      shift = k - blurDiam;

      // Basic edge clamp
      source = dest + shift;
      if (x+shift <= 0 || x+shift >= width) { source = dest; }

      // Read source pixel through kernel matrix from pixels[array]
      sourceR = red  (pixels[ source ]);
      sourceG = green(pixels[ source ]);
      sourceB = blue (pixels[ source ]);

      // Combine source and destination pixels with Gaussian Weight
      destR = red  (gaussPassX[dest]) + (sourceR * kernel[k]) /gaussSum;
      destG = green(gaussPassX[dest]) + (sourceG * kernel[k]) /gaussSum;
      destB = blue (gaussPassX[dest]) + (sourceB * kernel[k]) /gaussSum;

      // Store color in X pass array
      gaussPassX[dest] = color(destR, destG, destB);
    }
  }
  }

  // Do Vertical Pass
  for (int x = 0; x < height; x++){
  for (int y = 0; y < width; y++){
    dest = y*width+x;

    // Iterate through kernel
    for (int k=0; k < gaussWidth; k++){

      // Get pixel-shift (pixel dist between dest and source)
      shift = k - blurDiam;

      // Basic edge clamp
      source = dest + (shift*height);
      if (y+shift <= 0 || y+shift >= height) { source = dest; }

      // Read source pixel through kernel matrix from pass X
      sourceR = red  (gaussPassX[ source ]);
      sourceG = green(gaussPassX[ source ]);
      sourceB = blue (gaussPassX[ source ]);

      // Combine source and destination pixels with Gaussian Weight
      destR = red  (gaussPassY[dest]) + (sourceR * kernel[k]) /gaussSum;
      destG = green(gaussPassY[dest]) + (sourceG * kernel[k]) /gaussSum;
      destB = blue (gaussPassY[dest]) + (sourceB * kernel[k]) /gaussSum;

      // Store color in Y pass array
      gaussPassY[dest] = color(destR, destG, destB);
    }
  }
```

```
149.      }
150.
151.      // Copy Gaussian Y buffer to pixels[array]
152.      for (int pixel = 0; pixel < pixels.length; pixel++){
153.          pixels[pixel] = color( gaussPassY[pixel] );
154.      }
155.
156.      // Update the Canvas with the pixel[array]
157.      updatePixels();
158.
159.      int timeEndMS = (millis() - timeStartMS) / 1000;
160.      // println("Blur took: " + timeEndMS + " seconds.");
161.  }
```

Use this as you wish. Links are always appreciated.

- Index
- |
- J.Resig's PJS
- |
- Processing.js Google Group
- |
- My PJS Fork
- |
- RSS Feed
- |
- The Burst Engine