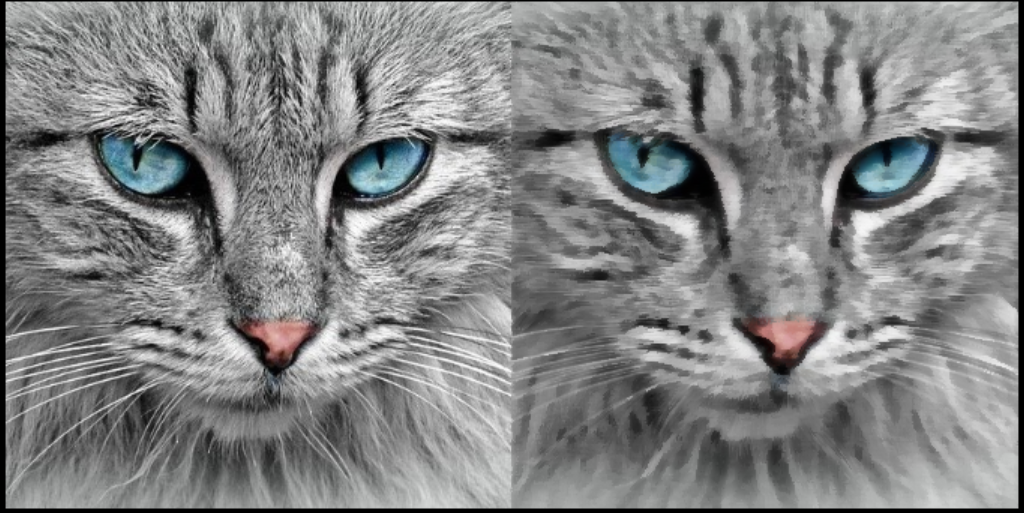
Blurring in GLSL

**McIntyre, Lewis, S2028246, Software Dev**



**Declaration**

*I confirm that the code contained in this file (other than that provided or authorized) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

*Signature. Lewis McIntyre*

[Graphics Programming (ACG) M3I625657-22-B](https://blackboard.gcal.ac.uk/webapps/blackboard/execute/courseMain?course_id=_115514_1)

2023

Table of Contents

[1. Introduction 3](#_Toc135057405)

[2. Box Blur 4](#_Toc135057406)

[3. Gaussian Blur 7](#_Toc135057407)

[4. Kuwahara Filter 13](#_Toc135057408)

[5. References 22](#_Toc135057409)

[6. Figures 22](#_Toc135057410)

# Introduction

There are a multitude of ways to blur an image, this can include a box, bilateral or radial filter. A fraction of these hold uses to this day, one of note being the Kuwahara filter. First used by Michiyoshi Kuwahara in the late 1970s for imagery of the heart, it then went on to gain numerous applications including edge detection, image processing and more recently tectonic plate modelling. (Kafadar, 2022) This document aims to display an implementation of the traditional Kuwahara filter, as well as a Box and Gaussian Blur. As you continue through this document, it will become clear the similarities within these techniques.

The vertex shader will stay the same throughout each of these blurring techniques, it is relatively straightforward and is as follows:



# Box Blur

The Box Blur is an incredibly useful tool, it is simple in design and is easy to setup. It’s use case comes generally from noise reduction, as can be seen in Figure 1. The box blur is not edge-preserving, meaning the final image will be unintelligible at higher blur levels. Later in this document, an edge-preserving technique named the ‘Gaussian Blur’ will be considered and compared to this technique.

A picture containing human face, screenshot, person, black and white

Description automatically generated

Box Blur Noise Reduction (9x9 Kernel) (Figure 1)

The title ‘*Box Blur*’ generalizes exactly how it operates. The implementation starts by going through every fragment within the image. At each fragment, a box is drawn of any size around it, in the case of Figure 2, a 5x5 box. The current fragments colour is then calculated by averaging every colour within this box.

A picture containing square, rectangle, symmetry, window

Description automatically generated

Box Blur (Figure 2)

Within GLSL, this will look as such. The first variables needed include a sampler2D to hold the image data, a vec2 to hold the texture coordinates, and finally an integer to hold the blur factor for the grid.



The first method to be called is main(), which holds the functionality to send out the current fragments colour. Since the shader is relatively simple, it will be calculated entirely in its own method. This will be called CalculateBoxBlur() and will return a vec4 holding its colour.



Kernel size defines the size of the box around the current fragment, and in the case of a box blur is directly proportional to the blur factor. A validation check should be taken to ensure the kernel size is above zero, which would suggest zero blurring. This is also because a negative blur cannot be applied. The textures size is then initialized as well as a vec4 to hold the average colour.



A grid will then need to be created, its size found using the formula, 2k+1, where k is the kernel size. This formula ensures the kernel size is always odd, and therefore has a center point, which will be the current fragment.



Within this grid, the texture coordinates of the current point in the grid can be calculated, and then the colour of that point should be added to the ‘*averageColour*’ variable.



The average colour can then be found using the mean formula which can be seen in Figure 3. Where is the mean, x is the sum of data points and N is the number of data points.

The Mean Formula (Figure 3)

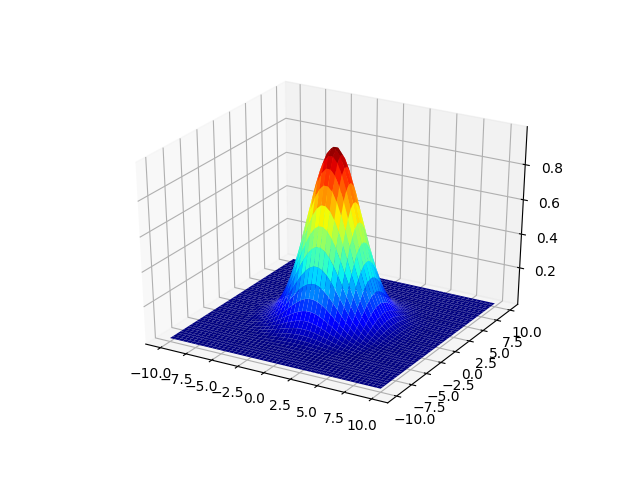
Since the sum of all data points has already been found the next step is to calculate the number of data points collected. This can be done by taking the number of iterations the loop goes through, ‘*2k+1*’, and multiplying it by itself for the number of loops used, which is two. The final formula is ‘*(2k+1)^2*’.

The average colour is then returned and applied to the fragment.



# Gaussian Blur

The Gaussian Blur is a blurring technique like the Box Blur however, the difference is in the weighting factors used. The Box Blur does not include any weighting, meaning every point in its grid has the same impact on the final output. On the other hand, the Gaussian Blur uses the Gaussian Distribution created by Carl Gauss in 1809 as a weighting factor for each point. This is the main issue with the Box Blur and a large factor in the Gaussian Blurs dominance in photo-editing software. An example of the Gaussian Blurs weighting factor can be seen in in Figure 4. The benefits of the Gaussian Blur can be shown by comparing the Box Blur. In Figure 5 it can be seen that, even though they have the same kernel size, the Gaussian Blur has a much greater effect on edge-preservation as the center of the Box Blur is basically unintelligible.



Gaussian Distribution (Figure 4)

A close-up of a person's face

Description automatically generated with low confidence

Box Blur vs Gaussian Blur, 25x25 kernel (Figure 5)

To implement this in GLSL, the Blur Shader implemented earlier should be used with a few adjustments. First all variable names and functions with relevance to the Box Blur have been converted to Gaussian Blur. The averageColour has been renamed to the weightedAverageColour and the total weight calculation has been removed for now. It is now as follows:



Next, a function to calculate the Gaussian weighting for each colour in the grid should be created. To do this a new function called CalculateGaussianWeight should be created taking in a vec2 holding the position of each object, as well as a float holding the standard deviation, which is usually expressed as the Greek letter sigma (σ). The standard deviation acts as an internal blurring factor, the higher the standard deviation resulting in a lower deviation from the center fragment.



Next the Gaussian Function should be implemented, which one dimensionally is:

1D Gaussian Function (Figure 6)

And therefore, two dimensionally is:

2D Gaussian Function (Figure 7)

Where σ is the standard deviation, x and y are the relative positions away from the center fragment, and the mean is zero.

This can be represented in GLSL as the following.



This function will then be called on every point in the generated grid to calculate its current weight. The current position in the for loop will be passed through, as well half the kernel size to keep the blurring factor consistent. Each weight will be added to the ‘*totalWeight*’ variable, so that the average can be calculated later. The weight is then applied directly to the current colour and added to the sum. This is as follows.



Since the average calculation has already been previously implemented, the Gaussian Blur should now function as intended. The Gaussian Standard Deviation calculation can be adjusted to anything as fit, a common implementation includes parsing a partial amount of the image and using its standard deviation to create a more useful shader.

# Kuwahara Filter

As previously mentioned, the Kuwahara Filter was originally created to reduce noise in medical imagery, however it has a secondary application in creating a pointillism-like painting aesthetic on 2D images. (See Figure 8)

A picture containing cloud, outdoor, water, sky

Description automatically generated

Kuwahara Filter 9x9 (Figure 8)

The Kuwahara Filter functions similar to the Box Blur by creating a grid, however the grid is split into four corners as seen in Figure 9. Next the average colour and standard deviation is found for each of these boxes, and the current fragment is replaced by the box with the lowest standard deviations average colour.

A picture containing rectangle, square, design, window

Description automatically generated

Kuwahara Filter (Figure 9)

This can be implemented in GLSL by using our Box Blur from earlier, however variable names such as BoxBlurKernel should be changed to KuwaharaKernel, as well the ‘*averageColour*’ variable. All of these changes are included in the following starter:



Firstly, since it is not one average colour that is being tracked, but four, an array of four colours should be declared. They can be in any order, however for the purposes of this shader, for 0-3 they are Bottom Left, Bottom Right, Top Left, and Top Right respectively. It should also have a separate array holding the count of each point in each sector, this will follow the same structure as before.

 we

Next, the average colour of each sector should be found, to do this, the same steps as before should be followed as well as using the equation as seen in Figure 3. However, it must only be carried out in each relevant sector. For example, in Figure 10, everything below zero in both the X and the Y, is in the bottom left square, therefore the sum and counting of any point in that square should be added to its relevant part of the array. The same principles can be applied on every point in the grid to find it relevant sector.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 |  |  |  |  |  |
| 1 |  |  |  |  |  |
| 0 |  |  | C |  |  |
| -1 |  |  |  |  |  |
| -2 |  |  |  |  |  |
|  | -2 | -1 | 0 | 1 | 2 |

Kuwahara Grid Example C=Center (Figure 10)

This code is long and tedious and as such, some code will be replaced with a TODO.



The averages should then be used with the equation found in Figure 3 using the counters as a weighting factor.



The standard deviation of each sector should then be found, using the equation found in Figure 11.

Standard Deviation Formula (Figure 11)

Where σ is the standard deviation, μ is the mean, x is a singular data point, and N is the number of datapoints.

To do this, an array of standard deviations must first be initialized.



Then, the grid should be generated and the same setup as the earlier loop should be added. The following code has been left undone with TODO’s as to save space and simplify its structure.



To find the standard deviation, a singular value must be found to represent each colour. This has been done by finding the magnitude of the colour using the dot product of itself. An example of adding this magnitude to the standard deviation is as follows. In this case, the index of the bottom left sector would be zero.



The standard deviation is then calculated by dividing each value by its counter.



Now that the standard deviation and average colour of each sector has been found, the lowest standard deviation sector can be chosen, and its relevant average colour will be used and returned as the final colour.



This filter is displayed in Figures 12, 13 & 14 with its kernel size increasing per image.

A picture containing x-ray film, black, monochrome, black and white

Description automatically generated

Kuwahara Filter, 1x1, 9x9, 17x17 and 33x33 Kernel Sizes (Figure 12)

A collage of a cat with blue eyes

Description automatically generated with medium confidence

Kuwahara Filter, 1x1, 9x9, 17x17 and 33x33 Kernel Sizes (Figure 13)

A picture containing panorama, sky, cloud, water

Description automatically generated

Kuwahara Filter, 1x1, 9x9, 17x17 and 33x33 Kernel Sizes (Figure 14)

# 5. References

Acerola. (2022, November 4). *This is the Kuwahara Filter: Acerola, Youtube*. Retrieved from Youtube Web site: https://www.youtube.com

Kafadar, O. (2022). Applications of the Kuwahara and Gaussian filters on potential field data. *Journal of Applied Geophysics*, 198-205.

Lettier, D. (2023, May 15). *3D Game Shaders for Beginners*. Retrieved from Lettier GitHub: https://lettier.github.io/

# 6. Figures

1. Image of Box Blur Noise Reduction 9x9: Image sourced from Peter on Boofcv.org.
2. Image of Box Blur Grid
3. Equation of the Mean Formula
4. Image of 3D Gaussian Distribution: Image sourced from Eduardo Mayo on StackOverflow.
5. Image of Gaussian Blur compared to Box Blur 25x25 Kernel: Image sourced from Peter on Boofcv.org.
6. Equation of a 1D Gaussian Function
7. Equation of a 2D Gaussian Function
8. Image of a Kuwahara Filter applied to an image 8x8 Kernel: Image sourced from Adam Marikar on unspash.com.
9. Image of the Corners of a Kuwahara Filter.
10. Table of an Example Grid for a Kuwahara Filter
11. Equation for the Standard Deviation
12. Kuwahara Filter applied to an image at Kernel Size 1x1, 9x9, 17x17 and 33x33: Image sourced from Peter on Boofcv.org.
13. Kuwahara Filter applied to an image at Kernel Size 1x1, 9x9, 17x17 and 33x33: Image sourced from Yeataro on TD-Anisotropic-Kuwahara GitHub.
14. Kuwahara Filter applied to an image at Kernel Size 1x1, 9x9, 17x17 and 33x33: Image sourced from Adam Marikar on unspash.com.