Subpixel level Pathtracing

How considering subpixels can increase the perceived resolution of a pathtracer

Jan C. de Munck

Supervisor(s): Elmar Eisemann, Michael Weinmann, Christoph Peters

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 20, 2025

Name of the student: Jan C. de Munck
Final project course: CSE3000 Research Project
Thesis committee: Michael Weinmann, Christoph Peters, Georgios Smaragdakis

Abstract

In this paper, we propose techniques to increase the effective resolution of a pathtracer without changing the resolution of the screen for which it is rendered. These make use of the fact that displays are made up of pixels, each of which is made up of three distinct subpixels. By setting luminance values for subpixels rather than colour values for a full pixel, we can create a higher fidelity image. This is already used in rendering text, but has not yet been applied to a pathtracer. It is done in three distinct ways; firstly we send three rays per pixel, each centred at a subpixel to obtain a value for every subpixel, the second method is using a filter used for downscaling text. Lastly, we implement a technique that makes use of the aforementioned filter without increasing the number of samples needed. By sampling a random subpixels each frame for each pixel and combining neighbouring pixels proportional to the filter. We find that these methods are quite effective at the goal of increasing perceived resolution in high-contrast scenes.

1 Introduction

Having a high resolution is a desirable quality of a pathtraced image; unfortunately the resolution of a display is fixed, but what if there is a way to increase their effective resolution? Physical displays are made up of many pixels, and each pixel is made up of several subpixels. LCDs commonly use a striped RGB pattern where each pixel is made up of a red, green, and blue subpixel in that order from left to right. We can exploit the fact that each subpixel is individually addressable to effectively triple the horizontal resolution of a display. The increased horizontal resolution does come at the cost of colour accuracy, however this is often a worthwhile trade-off since human eyes are more sensitive to luminance than colour [1].

Subpixel level rendering is already used, primarily in rendering text, like Microsoft's ClearType [2]. However it is not yet used in 3D pathtracers. While those could also benefit from the increased resolution. In this paper, three techniques will be implemented and evaluated. The first one traces three rays per pixel rather than one. Each ray is centred at a subpixel and is only responsible for the colour of that subpixel. Intuitively it seems like this would be very accurate as each subpixel is exactly the brightness the centre of that subpixel should be, however, it does not take the colour artefacts into account at all. The second technique is to render an image at triple the width and downscale it using the optimal filter described by John C. Platt [3]. This filter takes the way people perceive colour into account, and therefore should have less noticeable colour fringing. Both of these techniques require tracing three times more rays than a normal renderer for an image of the same resolution. The third technique traces one ray per pixel, but offsets this to be centred at a random subpixel, rather than always being centred at the full pixel. Then it combines neighbouring samples using the same filter as the second technique.

We show that all these techniques outperform traditional anti-aliasing, with the full filter being the best, followed by the built-in filter with random offsets, which in turn is better than the per-subpixel tracer. In terms of speed the full filter and per-subpixel pathtracer impose a massive performance penalty as they require tracing three times as many rays. So the built-in filter with random offsets provides a good balance between a low error and a fast execution.

Related works

This paper relies heavily on John C. Platt's paper *"Optimal Filtering for Patterned Displays"* [3]. Which is the basis of Microsoft's ClearType. Here, the optimal filter for downscaling an image to make use of subpixels is described. It also introduces a simple error function that is much easier to minimize than half-toning procedures requiring error loops, as was done previously [4; 5; 6]. These older methods require the repeated evaluation in a loop, whereas Platt's method is a linear filter. We make use of this error function to evaluate all methods. The filter itself is also one of the evaluated techniques, and serves as the basis for another.

For more detail around the human perception of colour *"Foundations of vision"* by Brian A. Wandell was conducted. Specifically, chapter 9 *"color"* [1] goes into human colour perception as well as different ways of organising colours. This includes an explanation of opponent colour space. In opponent colour space we have a brightness, a red-green, and a blue-yellow channel, rather than the red, green, and blue channels mostly used by computers. This opponent colour space closely resembles human perception of colour, therefore it is useful to consider the error in this colourspace with proper weighting of the channels.

While pathtracing research was consulted for the implementation, the details of the pathtracer are not relevant for the novel subpixel-level logic.

2 Methodology

We will now describe the problem more precisely. After that we will go into the methodology of the evaluation of the different techniques.

Anti-aliasing

When the pixels on a screen are large enough to make out individually, what should be smooth lines will appear as jagged edges. This is known as aliasing, and anti-aliasing techniques to mitigate this exist, but none that make use of subpixel-level logic are used in pathtracers. Conventional anti-aliasing techniques work by averaging many samples taken within one pixel, that way the jagged edges become gradients and the aliasing becomes less noticeable. This does not increase the resolution of the display, while it is possible to effectively triple the resolution of many LCDs.

By leveraging the fact that each pixel is made up of three distinct subpixels, and that human vision is much more sensitive to luminance than colour [1], we propose three methods that effectively triple the display resolution, at the cost of slight colour fringing.

Evaluation

The techniques will be evaluated using the error function minimized by the optimal filter [3], this function compares the image to a higher resolution image by first converting both into opponent colourspace, which is how people perceive colour [1], and then weighting it such that low-frequency changes, and the brightness channel get weighted higher than high-frequency changes, and red-green or blue-yellow channels. The goal is to minimize this error.

For the evaluation, we produce six images, three that are made by the new techniques, one that uses conventional anti-aliasing, one reference that does not use any subpixel-level logic or anti-aliasing and is of the same size as the ones being tested, and one that does not use any subpixel-level logic or anti-aliasing and is three times the width. The extra wide image is used in the error function to compare the techniques.

3 The techniques in detail

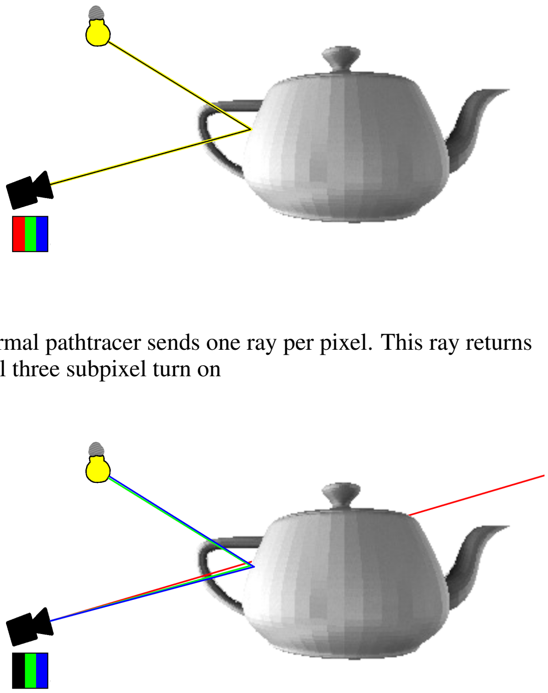
First, we develop a CPU pathtracer capable of rendering an image. This will be adapted for the subpixel level rendering techniques.

Per subpixel pathtracer

The first approach is to trace rays per subpixel, without doing any sort of anti-aliasing. This is a relatively straightforward procedure. At the start of the program, when initializing the camera, the width is tripled. The field of view is not changed so the same part of the scene is visible. This means that the density of rays will triple. When writing the image (either to screen or to disk) three neighbouring pixels are combined into a single pixel, each contributing one colour to one subpixel. Figure 1 illustrates how sending three rays per pixel can achieve more detail. A more precise algorithm is outlined in pseudocode below.

```
$colours \leftarrow pathtrace()$ ▷ pathtrace will return a 2D array
of colours; each colour is a vector of length three with the
RGB values.
for $y \leftarrow$ 0 to height do
    for $x \leftarrow$ 0 to outputWidth do
        for $c \leftarrow$ 0 to 3 do
            $outColours[y][x][c] \leftarrow colours[y][3x + c][c]$
        end for
    end for
end for
```

This way each subpixel is exactly the value that the centre of that subpixel should be. However, this still leaves aliasing, just at a smaller scale, and the colour fringing it introduces is in no way measured against human perception, can we do better?



(a) The normal pathtracer sends one ray per pixel. This ray returns white so all three subpixel turn on

(b) The per-subpixel pathtracer has three rays per pixel; The red one misses while the green and blue both return white, so only the green and blue subpixels turn on

Figure 1: An illustration of the per-subpixel pathtracer

Optimal filter

On top of the subpixel level pathtracer, we have also implemented the optimal filter described in [3]. In that work they have γ_{kd} as the starting image, with k as the position in pixels and d as the colour channel, and α_k as the output with k as the position in subpixels. They find the following objective function:

$$2\sum_{c,n,j} W_{cn}(\alpha_j M_{cj} - \sum_{d} C_{cd}\gamma_{jd})M_{ck}\cos\left(\frac{2\pi(k-j)n}{N}\right) = 0 \tag{1}$$

This can be rewritten as:

$$\sum_{c,n,j} W_{cn}M_{cj}M_{ck}\cos\left(\frac{2\pi(k-j)n}{N}\right)\alpha_j = \sum_{c,n,j} W_{cn}M_{ck}\cos\left(\frac{2\pi(k-j)n}{N}\right)\sum_{d} C_{cd}\gamma_{jd} \tag{2}$$

Here N is the width of the input image in pixels, C_{cd} is a 3x3 matrix that transforms γ_{jd} into opponent colour space, similarly M_{ck} is a 3xN matrix that does the same for α_k. W_{cn} is a low-pass weighting model that attributes more weight to low frequency changes and the brightness channel, while attributing less weight to high frequency changes and the red-green or blue-yellow channels. A precise definition of all identifiers can be found in the original paper [3].

Applying a little algebra shows that (2) is equivalent to solving a linear system $Ax = b$ with:

$$A_{kj} = \sum_{c,n} W_{cn} M_{cj} M_{ck} \cos\left(\frac{2\pi(k-j)n}{N}\right) \tag{3}$$

$$b'_{j,3k+d} = \sum_{c,n} W_{cn} C_{cd} M_{ck} \cos\left(\frac{2\pi(k-j)n}{N}\right) \tag{4}$$

$$\gamma'_{3j+d} = \gamma_{jd} \tag{5}$$

$$b = b'\gamma' \tag{6}$$

Now γ' is a flattened version of γ, and α_k is the solution to $Ax = b'\gamma'$, this can be rewritten as $\alpha_k = A^{-1}b'\gamma'$. Applying the filter is now just a single matrix vector multiplication, per row.

The matrix $(A^{-1}b')$ can be seen as 9 individual filters. Each row is responsible for one subpixel in the output image. Row i is responsible for a red subpixel iff $i\%3 = 0$, a green subpixel iff $i\%3 = 1$ and a blue subpixel iff $i\%3 = 2$. Since the filter applied is the same everywhere, all rows responsible for the same colour are identical, apart from being offset to be centred at the correct subpixel. Each column of the matrix corresponds to one input colour with column j being a weight for a red subpixel of the input iff $j\%3 = 0$ and green and blue correspond to $j\%3 = 1$ and $j\%3 = 2$ respectively. Now the 9 filters can be extracted as follows:

```
$RGB\_R \leftarrow Matrix.row(0)$
$RGB\_G \leftarrow Matrix.row(1)$
$RGB\_B \leftarrow Matrix.row(2)$
for $i \leftarrow 0$ to $N$ do
    $R\_R[i] \leftarrow RGB\_R[(3 * i) + 0]$
    $G\_R[i] \leftarrow RGB\_R[(3 * i) + 1]$
    $B\_R[i] \leftarrow RGB\_R[(3 * i) + 2]$
    $R\_G[i] \leftarrow RGB\_G[(3 * i) + 0]$
    $B\_G[i] \leftarrow RGB\_G[(3 * i) + 1]$
    $G\_G[i] \leftarrow RGB\_G[(3 * i) + 2]$
    $R\_B[i] \leftarrow RGB\_B[(3 * i) + 0]$
    $G\_B[i] \leftarrow RGB\_B[(3 * i) + 1]$
    $B\_B[i] \leftarrow RGB\_B[(3 * i) + 2]$
end for
```

Now R_R is the filter from red to red, R_G is from red to green, etc. The filters can be seen in Figure 2. The reason to extract these filters rather than keeping it as a matrix multiplication is to be able to compute the filter application in the frequency domain, which is much faster. The matrix multiplication is equivalent to the following convolution:

$$y[n] = \sum_{k} x[k]h[(k-n)\%N] \tag{7}$$

Where y is the output, x is the input, h is the filter, and N is the width. This is slightly different from a normal convolution, which works like so:

$$y[n] = \sum_{k} x[k]h[(n-k)\%N] \tag{8}$$

So in order to keep the frequency-domain filter application equivalent as the matrix multiplication, we must first reverse the filters, as well as circularly shifting them so that they are centred on index 0. The following shows the computation of the discrete Fourier transforms of all filters:

```
    ▷ Get the Discrete Fourier Transforms of the filters, this has to be done once.
$filters \leftarrow [R\_R, G\_R, B\_R, R\_G, G\_G, B\_G, R\_B, G\_B, B\_B]$
for $i \leftarrow 0$ to $9$ do
    $current \leftarrow filters[i]$
    $current.shift(-(floor(i/3) + 1))$
    $current.reverse()$
    $filter\_dfts[i] \leftarrow dft(current)$    ▷ $dft()$ computes the discrete Fourier transform
end for
```

Since the filters do not rely on the input image, only on the width, these dfts can be stored and reused whenever the filters are applied to a different image of the same width.
Below is the application of the filters assuming that all filter dfts are precomputed.

```
    ▷ Applying the filters, this has to be done once for every row in the image.
for $i \leftarrow 0$ to $N$ do
    $In\_Red[i] \leftarrow In[3 * i + 0]$
    $In\_Green[i] \leftarrow In[3 * i + 1]$
    $In\_Blue[i] \leftarrow In[3 * i + 2]$
end for
$In\_Red\_dft \leftarrow dft(In\_Red)$
$In\_Green\_dft \leftarrow dft(In\_Green)$
$In\_Blue\_dft \leftarrow dft(In\_Blue)$
for $i \leftarrow 0$ to $N$ do
    $Out\_Red\_dft[i] \leftarrow R\_R\_dft[i] * In\_Red\_dft[i]$
    $Out\_Red\_dft[i] += G\_R\_dft[i] * In\_Green\_dft[i]$
    $Out\_Red\_dft[i] += B\_R\_dft[i] * In\_Blue\_dft[i]$
    $Out\_Green\_dft[i] \leftarrow R\_G\_dft[i] * In\_Red\_dft[i]$
    $Out\_Green\_dft[i] += G\_G\_dft[i] * In\_Green\_dft[i]$
    $Out\_Green\_dft[i] += B\_G\_dft[i] * In\_Blue\_dft[i]$
    $Out\_Blue\_dft[i] \leftarrow R\_B\_dft[i] * In\_Red\_dft[i]$
    $Out\_Blue\_dft[i] += G\_B\_dft[i] * In\_Green\_dft[i]$
    $Out\_Blue\_dft[i] += B\_B\_dft[i] * In\_Blue\_dft[i]$
end for
    ▷ $idft()$ computes the inverse discrete Fourier transform
$Out\_Red \leftarrow idft(Out\_Red\_dft)$
$Out\_Green \leftarrow idft(Out\_Green\_dft)$
$Out\_Blue \leftarrow idft(Out\_Blue\_dft)$
$RGB \leftarrow [Out\_Red, Out\_Green, Out\_Blue]$
for $i \leftarrow 0$ to $N$ do
    $Out[i] \leftarrow RGB[i\%3][i]$
end for
```

Built-in filter with random offsets

Lastly, an implementation of the filter within the pathtracer was made. The idea behind this is that the subpixel level pathtracer requires three times as many rays to be traced for the same size image, and similarly the filter downsizes an image by 3x horizontally, so it also needs an image to be rendered at 3x the width for its output to be the same size. This means that using either of these techniques will slow the pathtracer down by a factor of 3 at least (before considering overhead). But what if there were a way to get similar subpixel level precision while still only tracing one ray per pixel per frame? In the next technique we sample one ray per pixel, but we ran-

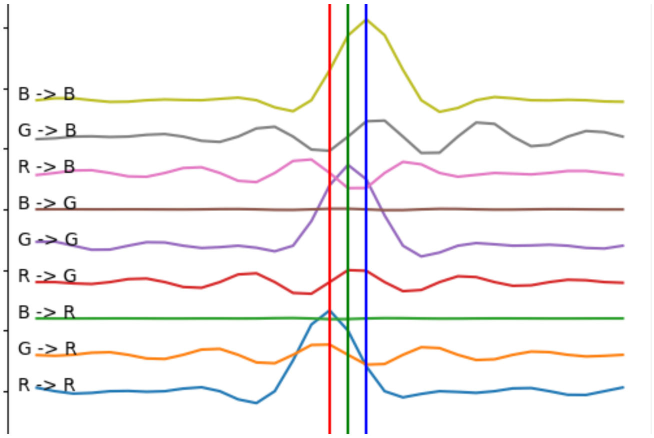


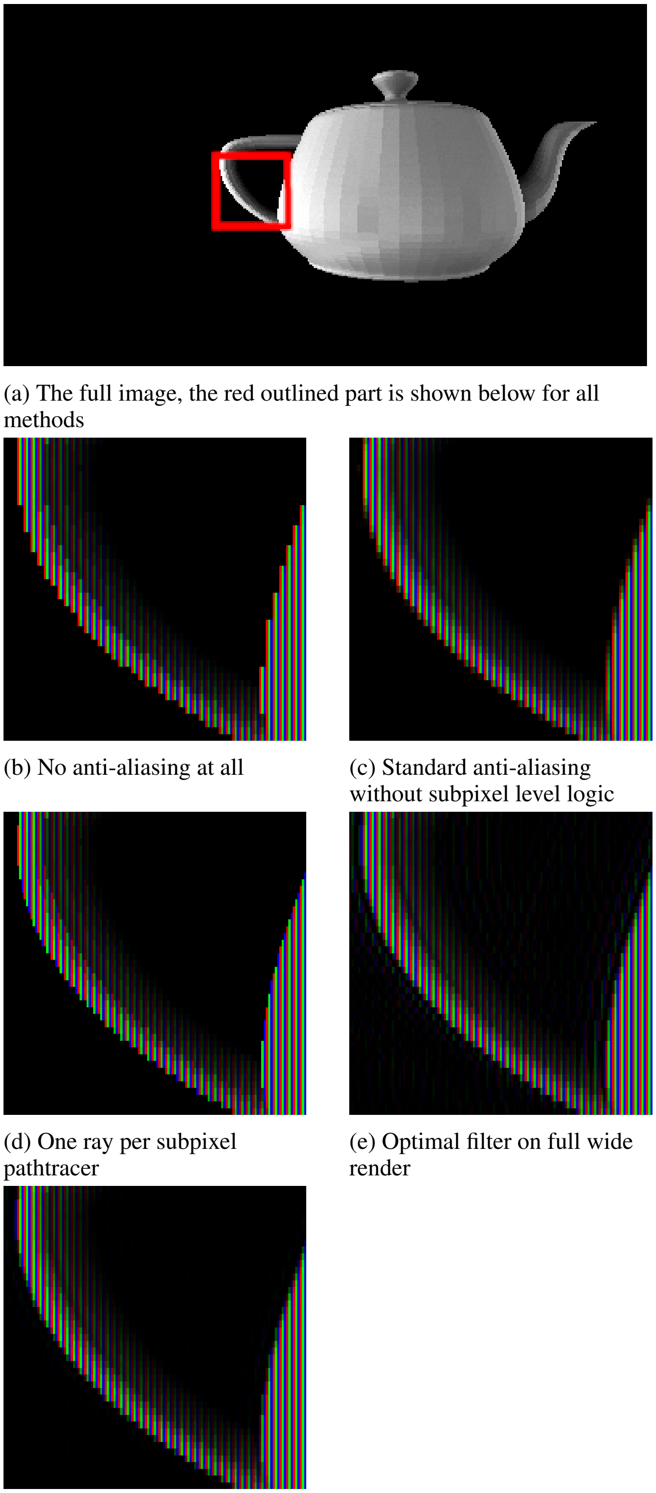
Figure 2: The 9 filters, each offset vertically so they are visible over each other. The vertical lines are the centers of the red, green, and blue subpixels.

domly offset it so that it is centred at a random subpixel. Then the final value of a subpixel is determined by combining the horizontal neighbours proportional to the filter. We make use of the previously computed filters, which can also be seen in Figure 2.

As the filters are now applied with three times fewer samples (one per pixel rather than one per subpixel) the result of filter application is multiplied by three to get the final result. Since for each pixel the subpixels which is sampled is randomized each frame, it is expected that eventually all subpixels are sampled a few times and a good image is produced.

In the pseudocode below $filtSize$ is the size of one of the filters, in our implementation we always used filters the size of trice the width of the image (so the size of the image when measured in subpixels) that way it is as similar as possible to the full filter. Smaller filters can be used for a slightly higher error, but faster computation.

```
$filters \leftarrow [R\_R, G\_R, B\_R, R\_G, G\_G, B\_G, R\_B, G\_B, B\_B]$
for $y \leftarrow 0$ to $height$ do
    for $x \leftarrow 0$ to $width$ do
        for $idx \leftarrow 0$ to $width$ do
            $offset \leftarrow rays[y * width + idx].offset$ ▷ get the subpixel that this ray was centred at, $-1$ means it was centered at the red subpixel, $0$ for green and $+1$ for blue
            $offset+ = 3(idx - x)$
            $offset = (offset + filtSize)\% filtSize$
            for $f \leftarrow 0$ to $9$ do
                $outColours[y][x][floor(f/3)]+ = colours[y][idx][f\%3] * filters[f][offset]$
            end for
        end for
        $outColours[y][x][0]* = 3$
        $outColours[y][x][1]* = 3$
        $outColours[y][x][2]* = 3$
    end for
end for
```


(a) The full image, the red outlined part is shown below for all methods

(b) No anti-aliasing at all

(c) Standard anti-aliasing without subpixel level logic

(d) One ray per subpixel pathtracer

(e) Optimal filter on full wide render

(f) Optimal filter in renderer with random offsets

Figure 3: Zoomed in versions of the different techniques. (printing introduces artefacts, best viewed on a screen rather than paper)

technique	Scene 1	Scene 2	Scene 3	Scene 4
Control	270.021	1763.13	2124.14	2279.34
Conventional anti aliasing	250.24	1349.65	2033.83	1858.76
Per-subpixel pathtracer	230.811	1261.36	2004.73	1737.18
Full filter	**190.538**	**857.625**	**1620.45**	**1199.39**
built-in filter with random offsets	213.369	1048.99	1846.92	1393.57

Table 1: RMSe of all methods across 4 scenes

technique	Scene 1	Scene 2	Scene 3	Scene 4
Control	105.952	103.766	105.465	103.137
Conventional anti aliasing	121.243	118.779	121.495	119.048
Per-subpixel pathtracer	312.947	312.111	321.859	316.299
Full filter	312.824*	311.408*	321.913*	315.107*
built-in filter with random offsets	149.241	146.728	150.084	147.32

Table 2: Time in ms for all methods across 4 scenes

4 Experimental Setup and Results

After the implementation of all techniques, they were tested by each rendering several scenes for enough frames to converge. These renders are then compared to a render 3x wider using the error function from [3]. As this function returns an error per row of the image, the RMS of the errors of all rows is taken to get one error value per image per technique. The implementation was made in c++. The pathtracer makes use of the tinyBVH library (https://github.com/jbikker/tinybvh), and the filter and error rely on the Eigen library (https://eigen.tuxfamily.org/index.php?title=Main_Page) for the Fourier transform, and linear algebra. The full implementation can be found at https://gitlab.ewi.tudelft.nl/cjppeters/realistic_rendering/-/tree/Jan_main?ref_type=heads. All tests were run with 400x225px images on an AMD Ryzen 5 5600H. This resolution was chosen because it is a large enough image to get a pleasing image, while being small enough to allow rendering for thousands of frames to eliminate noise. If too much noise is present, the noise will play a large role in the error, so the error would not be reflective of the quality of the subpixel-level anti-aliasing.

A tool was developed to better visualise the renders, as zooming in on the renders themselves will show pixels of the same size but different colours, rather than showing the increased fidelity achieved by the subpixel level logic. In Figure 3 the cropped output of this tool can be seen for all techniques. Some qualitative information can be inferred from these. Namely that while the optimal filter does minimize the error, and looks really good on the edge, it also introduces artefacts next to the edge. Some ghosting is visible next to the edge, which should show the black background. This is expected as we can see from the filters in Figure 2 that a subpixel's value is influenced by the value of pixels several pixels over.

Quantitative results

The RMS error for every scene can be seen in Table 1. The full filter consistently has the lowest error, followed by the built-in filter with random offsets. In third place comes the pathtracer that traces a ray for every subpixel, and conventional anti-aliasing does a little better than doing nothing.

Performance was also measured as a secondary metric. In Table 2 are the average times for generating a frame for every scene and every technique.
For the full filter this is just the mean time of generating one frame of the full-width image. Applying the filter took on average 32.857ms. This is measured separately because the filter only needs to be applied to the final accumulated frame. The longer you let the renderer converge before applying the filter, the smaller portion of the total time filter application will be.

5 Discussion

It is not surprising that the full filter has the lowest error; it is specifically designed to minimize the error, so was expected to do well. What is notable is how much better the full filter performs compared to the renderer which applies the filter to randomly sampled subpixels. The difference can also be seen in Figure 3, where applying the filter on the full-width image leaves a lot more ghosting than applying it with random subpixels. The randomness and having fewer samples mean that the built-in filter with random offsets will have a larger error than applying the filter once to a full-width image.

As expected, not doing any subpixel level logic or anti aliasing performs worst, followed by normal anti aliasing without considering subpixels. This shows that considering subpixel level logic is indeed a improvement over existing anti-aliasing rendering techniques. The simple subpixel level pathtracer scored right in the middle, still a big improvement over doing nothing, or conventional anti-aliasing, but not as good as the filter. This also makes a lot of sense, as it still has aliasing, just at a smaller scale, and the colour artefacts it produces are not controlled for by a weighting model like in the filter.

When it comes to performance, it is no surprise that the per-subpixel pathtracer and full filter application are approximately three times slower than the control, while conventional anti-aliasing and the built-in filter with random offsets have less impact on performance. But they do both slow the renderer down somewhat, a reason contributing to both conventional anti-aliasing and the built-in filter with random offsets is that for every frame for every pixel a new random offset is chosen and therefore the full ray needs to be traced again. In the methods without random offsets the rays from the camera to the scene are computed once and reused if the camera has not moved between frames (which it never does in the tests). In the case of the built-in filter the filter application itself is also responsible for some of the performance decrease.

6 Conclusions and Future Work

Our work has shown that subpixel-level logic can vastly improve the quality of pathtracers. For the best results, the optimal filter from John. C. Platt [3] is used. But this introduces a large performance penalty. A good balance between a low error and a fast execution can be achieved by using the filter within a pathtracer which samples random subpixels. Depending on whether or not the pathtracer needs to work in real-time, a developer can choose between these options.

Future work

In the future more research can be done in applying this logic to other subpixel layouts. This paper focused entirely on the striped RGB layout of most LCD displays, and this is easily adapted to BGR, vRGB, or vBGR but some other layouts [7] could prove more challenging. Layouts exist where not every pixel has the same arrangement of subpixels, or even where pixels share subpixels. Algorithms for these layouts would create images that have to be shown in a specific place on the screen, as moving it by a single pixel would mean an image pixel computed for arrangement A is now displayed on a physical pixel with arrangement B.

Another aspect future research can look into is doing user studies to see to what extent people actually prefer the images with a lower error. It is entirely possible that the error function shows large differences between images, while for people looking at a screen the differences are hard to notice. Finding this point of diminishing returns would be very valuable in considering which technique (if any) to implement. After all; the target audience of any graphics application is people, not any error function.

7 Responsible Research

Due to the nature of this paper the main concern regarding responsible research pertains to reproducibility. As all novel algorithms are described in the paper and the sourcecode is available at https://gitlab.ewi.tudelft.nl/cjppeters/realistic_rendering/-/tree/Jan_main?ref_type=heads, the research is highly reproducible. In picking scenes to test we made scenes with a black background and a bright object to create high-contrast areas, with a sharp division between black and bright areas. The anti-aliasing that is essentially performed is more useful in sharp lines than with gradients. In low-contrast scenarios there is not much aliasing present and therefore no need for anti-aliasing techniques. This means that our results will show a greater improvement over the baseline than the average case. In low-contrast scenes the baseline of doing no subpixel level logic or anti-aliasing will already have a small error, so no technique can meaningfully improve on it.

References

[1] Brian A Wandell. *Foundations of vision.*, chapter 9: Color. Sinauer Associates, 1995.

[2] Microsoft. Microsoft cleartype overview. online article, 2022. https://learn.microsoft.com/en-us/typography/cleartype/.

[3] John.C. Platt. Optimal filtering for patterned displays. *IEEE Signal Processing Letters*, 7(7):179–181, 2000.

[4] Bernd W. Kolpatzik and Charles A. Bouman. Optimized error diffusion for high quality image display. 1992.

[5] Jeffrey Mulligan and Albert Ahumada. Principled halftoning based on human vision models. 02 1970.

[6] David L. Neuhoff, Thrasyvoulos N. Pappas, and Nambi Seshadri. One-dimensional least-squares model-based halftoning. *Journal of the Optical Society of America A: Optics and Image Science, and Vision*, 14(8):1707–1723, August 1997.

[7] Agatha Mallet. Subpixel zoo: A catalog of subpixel geometry. online article. https://geometrian.com/resources/subpixelzoo/.