Efficient on-GPU blurring

Max Reitz

Abstract

Blurring a texture is a common operation in various algorithms. Using a Gaussian blur even with separation has linear complexity depending on the blur radius. This paper introduces an algorithm for approximating Gaussian blur with logarithmic complexity using an iterative approach.

Introduction

The simplest way of blurring an image is to down- and upsample it using appropriate filters. However, because information is lost in the process, the result differs strongly from a real Gaussian blur, even with complex scaling filters; generally, it is not acceptable.

Most blurring operations in computer graphics do not require the result to fulfill specific conditions other than "looking good" (which includes approximately circular blurs and brightness conservation). Gaussian kernels fulfill this conditions, therefore they are used as a quasi-standard and are perfect as a reference to compare against.

The Gaussian blur, as all blurs which should not exhibit artifacts as the ones seen from down- and upscaling, requires for each pixel to take all sourrounding ones into consideration. Therefore, there are multiple texture reads per pixel. To increase performance, one can either decrease the number of pixels or the number of texture reads. For a good blur, it is however not possible simply not to consider a significant amount of neighboring pixels. Also, if the blur radius is x pixels wide, neighbors up to that distance have to be sampled for each fragment. Larger blur radii therefore require more texture reads.

Decreasing the pixel count

A common way to speed up blurring is to perform it on low-resolution framebuffers with bilinear filtering. Since the blurred result has low contrast by nature, upscaling results in few artifacts. The artifacts resulting from downscaling on the other hand are simply blurred away. Therefore, for still images, this is a very easy way of dramatically increasing performance: Images with about a 16th of the original border size (that is, $\frac{1}{256}$ of the original pixel count) still look fine (generally, it seems to be related to the kernel size: $r = 7$ kernels are fine, $r = 3$ kernels are not).

However, this comes with a great disadvantage: If the input is offset a little, for instance because of a slow movement, the blurred result flickers. This is particularily visible for bloom blurs. In their case, even a fourth of the original border size may be too small; in extreme cases, nothing but the original image size may be satisfactory.

So, beware: This method looks great for still image examples; but for moving imagery, it may have a noticeable quality degradation. If it works, on the other hand, it dramatically increases performance and is very easy to employ.

Reducing the kernel size

Common approaches

Directly using a Gaussian kernel as a two-dimensional filter means having quadratic complexity regarding the blur radius. This is unacceptable.

Because the Gaussian kernel is separable, however, it is sufficient to apply two one-dimensional kernels consecutively; this results in linear complexity, or, to be more specific, in $4r - 2$ texture reads ($2r - 1$ per pass). This may be enough, but still means that a large blur radius (which may be very much desirable for e.g. bloom) is rather time-consuming.

Relying on built-in filtering

It is possible to use the GPU's built-in bilinear filtering to reduce the number of `texture()` calls in the shader code (because every `texture()` call then reads at least two texels for interpolation, this doesn't reduce the number of texel fetches; but since the GPU can interpolate the texels very fast, this still increases performance). The weight of two

neighboring pixels is then used to calculate a texture coordinate which automatically interpolates both.

The original code looks like this (for the horizontal blur shader):

```
/* ... */
+ texture(tex, vec2(tc.x + res * i
                    tc.y)) * w1
+ texture(tex, vec2(tc.x + res * (i + 1)
                    tc.y)) * w2
/* ... */
```

And the code using built-in bilinear interpolation then looks like this:

```
/* ... */
+ texture(tex,
          vec2(tc + res * (i +
               abs(w1 - w2) / (w1 + w2)),
               tc.y)) * (w1 + w2)
/* ... */
```

This way, the number of texture reads can be reduced by nearly half to $2r$ in total (r per pass). This is nice but still remains linear.

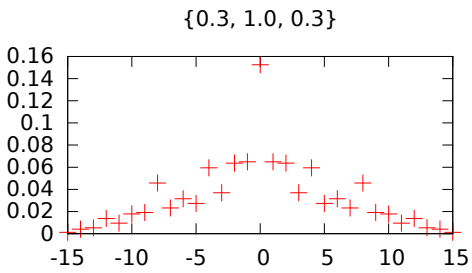
Leaving Gauss

As stated before, Gaussian kernels are mainly used because they look nice – not because they are actually required. Therefore, we are free to leave the Gaussian distribution and use whatever looks similar.

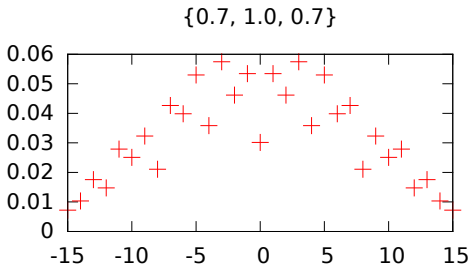
The way which is to be presented here results from the following idea: Instead of having a single pass (or at least one per direction), we can use multiple with exponentially decreasing step width. In every step, every fragment samples only two pixels (and itself) per direction. In the first step, those are very far away (approximately the desired Gauss radius). In the next step, the distance is halved; then it is halved again; and so on, until the directly neighboring pixels are sampled.

This obviously results in logarithmic complexity regarding the blur radius: $6 \log_2 r$ (3 texture reads per direction and pass). For instance, for a blur size of 128 pixels, this means 42 reads in contrast to 256 using bilinear filtering and a real Gauss kernel.

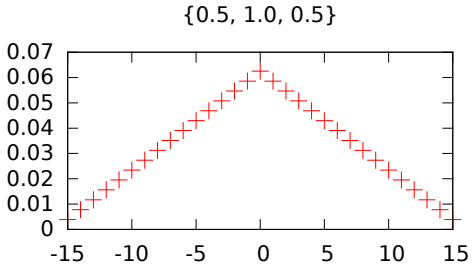
If one uses the same kernel for all passes, there are interferences which result in over-amplifications:



Or with higher weighting:

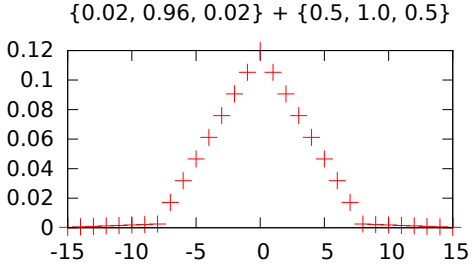


The only weight distribution which is free from such interferences is the "exponential" kernel $\{0.5, 1.0, 0.5\}$ (or $\{0.25, 0.5, 0.25\}$ when normalized):



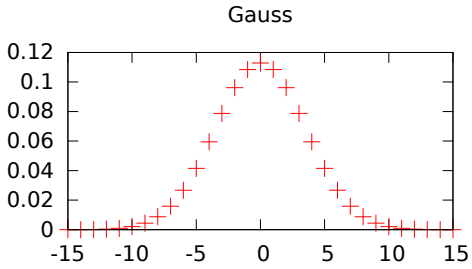
But the curve should be rather flat at the edges and more steep in the center.

A solution is to choose a very "hard" kernel in the first pass and the "exponential" in the rest:



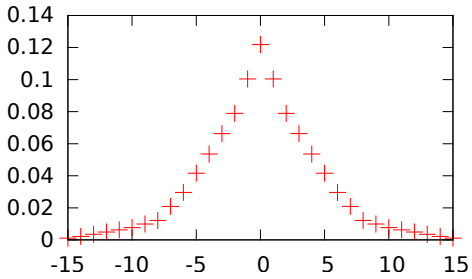
The initial kernel tunes the overall distribution, all of the following do something like a linear interpolation.

Although this does not look too good at first glance, compare it to a Gauss curve:



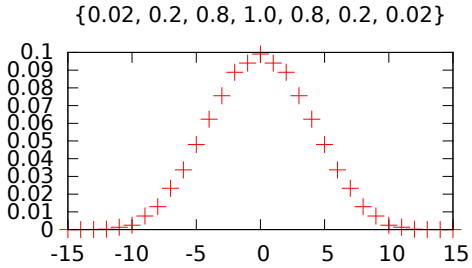
Both are actually rather similar; even more so in the visual appearance.

Other kernels with varying degree of "hardness" for each pass are possible, but they do not fundamentally improve the situation:



This one was specifically crafted for the required number of passes (4). Creating a similar one ad-hoc for another number of passes would not be trivial. Since it does not provide a fundamental improvement (see the examples below), there is no reason to use it.

Another way to improve the quality is to use a more fine-grained initial kernel to set the overall shape:



The complexity with such a kernel would be $6\log_2 r + 8$. The visual appearance does not improve much, though; most notably, it increases the apparent blur radius.

Built-in filtering

It is impossible here to use built-in filtering the way it can be used for real Gauss kernels. Since all but the very last kernel do not actually read adjacent texels, using offsets for the texture reads would not interpolate the two targeted texels but rather just use some texel(s) between both.

The only kernel which might make use of this filtering is the very last; the six texture reads per pixel could be decreased to four (and a single pass instead of one per direction). The total complexity would then be $6\log_2 r - 2$ or $6\log_2 r + 6$, depending on the initial kernel. All in all, it therefore does not make much of a difference.

Combining everything

The complete complexity of a blur operation is $n_w \cdot n_r$, where n_w is the number of writes (that is, the number of pixels) and n_r is the number of texture reads per pixel written.

n_w can and is commonly decreased by decreasing the size of the blur buffers using bilinear interpolation. For still images, it can be decreased by a factor of up to 256 (in some cases even more). For moving images however, it should be decreased by much less and sometimes cannot be decreased at all.

For two-dimensional blur kernels, $n_r \in O(r^2)$ (with r being the blur radius). The commonly employed Gaussian kernels are separable, therefore this can be decreased to $n_r \in O(r)$ ($n_r = 4r - 2$). Using built-in bilinear interpolation, it can be further decreased to $n_r = 2r$ (although the number of actual texture fetches remains the same and the complexity is still linear).

Using the iterative approach presented here, it is possible to achieve a Gauss approximation with $n_r \in O(\log r)$ ($n_r = 6\log_2 r$ or $n_r = 6\log_2 r + 8$ for a slightly better approximation). It is still very much possible to decrease n_w independently.

Since the iterative approach requires multiple passes, it has some overhead which cannot be captured by these terms (changing framebuffers, drawing a screen-filling quad, etc.). Therefore it is probably inappropriate for small blur radii where the complexity of both approaches is still about the same (even past the break-even point of $r > 8$). Probably the greatest disadvantage of the iterative approach is its inability to be easily adapted to arbitrary blur radii; the radius can only be doubled or halved. Adapting the initial blur kernel may help if another radius is absolutely required.

Visual comparison

For this document, an originally 1080p image has been scaled down to $\frac{1}{16}$ (480x270) so that details are more clearly visible.

Figure 1: Original image

Full images

Ordered ascending by the apparent blur radius:

Figure 2: Iterative (modified initial kernel), 4 passes, full image

Figure 3: Gauss, $r = 31$, full image

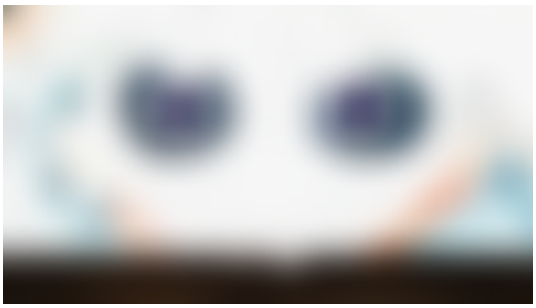


Figure 4: Iterative, 5 passes, full image

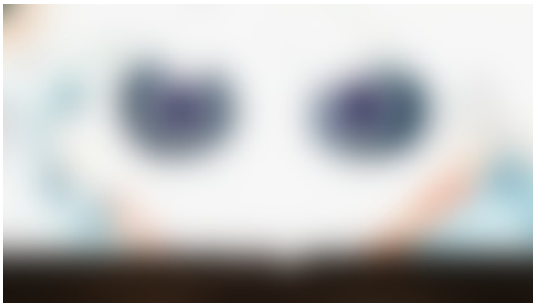


Figure 5: Gauss, $r = 50$, full image

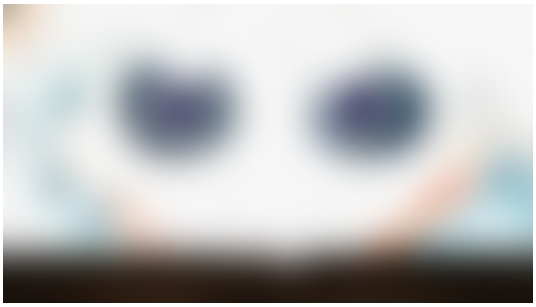


Figure 6: Modified iterative, 5 passes, full image

As you can see, the only visible difference is the apparent blur radius. Apart from that, there is no visible quality difference.

Downscaled images

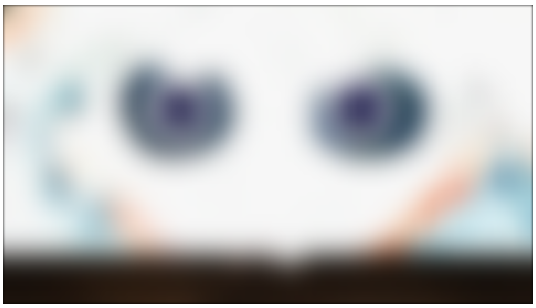


Figure 7: Gauss, $r = 7$, $\frac{1}{16}$ image area

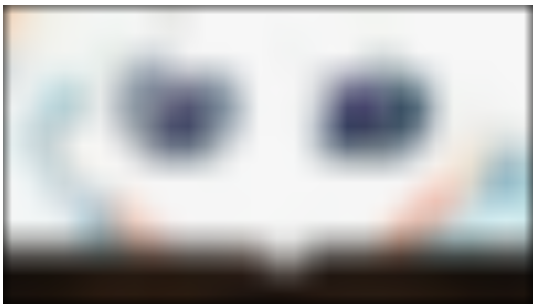


Figure 8: Gauss, $r = 3$, $\frac{1}{64}$ image area

The first image still looks good; the second one shows clear artifacts probably due to the small kernel size (a 1080p image still looks fine when blurred at $\frac{1}{256}$ of the original size with $r = 7$).

Downscaled movement

The following images are the amplified (both amplified equally) difference of the image simply blurred and the image moved down by two pixels, then blurred and then moved up again. They reveal the problem of movement flickering.

Figure 9: Movement difference for an image which was downscaled before blurring ($\frac{1}{16}$)

Figure 10: Movement difference for an image blurred with the original size

The second image exhibits some artifacts due to "foreign" pixels added at the top for moving it down. Other than that, only the bottom has some minor disturbances. The first image however has clear artifacts all over the image, especially along the borders of the original image.

Performance comparison

Gaussians were performed without built-in bilinear interpolation; however, take into consideration that the apparent blur radius of the iterative approach (especially the modified one) is larger than the Gaussian radius, so both errors should equal out.

AMD HD 7850

Full-size:

- 1920x1080, Gaussian ($r = 127$):
  399 ms ±959 µs
- 1920x1080, iterative (7 passes):
  2.72 ms ±139 µs
- 1920x1080, modified iterative (7 passes):
  2.88 ms ±130 µs

(I cannot explain the extreme value for the Gaussian blur; but it is reproducable, so I am willing to blame the driver)

Downscaled to $\frac{1}{16}$:

- 480x270, Gaussian ($r = 31$):
  893 µs ±108 µs
- 480x270, iterative (5 passes):
  656 µs ±137 µs
- 480x270, modified iterative (5 passes):
  712 µs ±134 µs

Downscaled to $\frac{1}{64}$:

- 240x135, Gaussian ($r = 15$):
  352 µs ±86.5 µs
- 240x135, iterative (4 passes):
  479 µs ±79.3 µs
- 240x135, modified iterative (4 passes):
  499 µs ±67.9 µs

Downscaled to $\frac{1}{256}$:

- 120x67, Gaussian ($r = 7$):
  221 µs ±82.4 µs
- 120x67, iterative (3 passes):
  378 µs ±86.8 µs
- 120x67, modified iterative (3 passes):
  383 µs ±81.9 µs

Intel HD 4000

Full-size:

- 1920x1080, Gaussian ($r = 127$):
  112 ms ±50.2 ms
- 1920x1080, iterative (7 passes):
  30.8 ±5.39 ms
- 1920x1080, modified iterative (7 passes):
  30.7 ms ±5.39 ms

Downscaled to $\frac{1}{16}$:

- 480x270, Gaussian ($r = 31$):
  5.42 ms ±90.5 µs
- 480x270, iterative (5 passes):
  2.64 ms ±1.11 ms
- 480x270, modified iterative (5 passes):
  2.77 ms ±1.17 ms

Downscaled to $\frac{1}{64}$:

- 240x135, Gaussian ($r = 15$):
  1.44 ms ±140 µs
- 240x135, iterative (4 passes):
  1.13 ms ±417 µs
- 240x135, modified iterative (4 passes):
  1.17 ms ±348 µs

Downscaled to $\frac{1}{256}$:

- 120x67, Gaussian ($r = 7$):
  882 µs ±367 µs
- 120x67, iterative (3 passes):
  791 µs ±429 µs
- 120x67, modified iterative (3 passes):
  814 µs ±422 µs

Conclusion

As one can see, the algorithm presented here is useful only for large blur radii ($r > 25$). For smaller radii (which are especially common on small-sized or downscaled textures), the overhead is too large and the well-known standard algorithm is simply fast enough.

On the other hand, for integrated GPUs without dedicated memory for which texture reads are especially costly, the iterative algorithm is faster even for small blur radii.