One thing that makes HDR rendering impressive is the bloom effect. In fact even if bloom is in a sense decoupled from HDR rendering, it is often confused with HDR. There are 3d engines out there that advertise bloom for HDR rendering, which is nonsense. You can have HDR rendering without bloom, and you can have bloom without HDR rendering. What makes the sweet, sweeter is the combination of the two. Actually if you present the average gamer HDR rendering without bloom, it will be hard for her to understand the difference between LDR and HDR rendering…

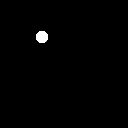
This means that you are going to need good blooming in your engine to really have that “WOW” thing coming out of peoples mouths. Before writing the [Sylphis3D](http://www.sylphis3d.com/) HDR rendering implementation I had read some articles about blooming but the results were never satisfactory. I’m going to present you here the method I used and that really makes a perfect bloom that is also faster to perform and uses less memory.

**The usual algorithm for bloom is this :**

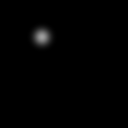
* Take the current render and downscale it to a reasonable size that is preferably a power of 2
* Apply a bright-pass filter on the image to keep only high luminance values
* Apply a Gaussian blur filter of small kernel size several times to get a good blur
* Additively blend the resulting texture on the screen

The above algorithm is what you do in general for bloom but has a flaw. The flaw is in the blurring step. This blurring is done by applying a Gaussian blur filter. This Gaussian filter is decomposed to two passes: one horizontal and one vertical. This way the samples required for the filter is reduced. I’m not going to get into this since I consider this as something trivial.

The problem is the size of the kernel of this filter. Since we are doing this in real-time and on the GPU, we can’t have as many samples as we like. This results in small kernels that create very small blur. The common solution to this problem is to repeat the blur step several times in order to expand the blur.



The unblurred   
bloom texture

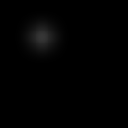


After 5 Gaussian  
blurs of 5×5 kernel

In the above images we can see the texture that results after the bright-pass filter of the left, and on the right we see the blurred image of it after applying a Gaussian blur with a 5×5 kernel 5 times. This bloom texture is not what I consider a good result. The reasons are:

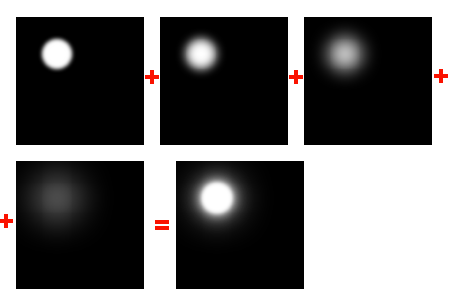
* **The image lost its intensity**: The several blurs resulted in a fading of the luminance.
* **The size of the blur is very small** : The resulting bloom is small and constrained at only a few pixels around the bright spots, when a very bright spot should give a wide glare and bloom.

The first that comes to mind is to solve this by increasing the kernel size.



After one Gaussian  
blur of 21×21 kernel

On the image to the left we can see the results of applying a Gaussian blur of 21×21 kernel size. The glare now has bigger and acceptable size, but the intensity is faded again. What we need is to keep some *sharpness* of the smaller kernels but also have the wide spread of the big kernels. We will do this by adding several blurred images of various kernel sizes.



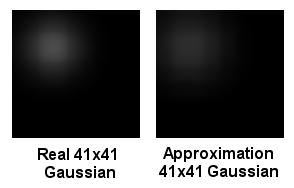
Adding multiple Gaussian  
filters

In the image above we can see the effect of 4 Gaussian blur filters of kernel sizes 5×5, 11×11, 21×21, 41×41 along with the result we get by adding them up. The result look as expected… very good. This is what we need.

At the moment you are probably starting to wonder: *how are we going to do a 41×41 kernel in real-time, on todays hardware?!* The answer comes next…

**Implementation**

The solution is as always: **approximation**. I will try to invert the problem. Think of it like this: I need to double the radius of the Gaussian blur kernel… wouldn’t that be like applying the same Gaussian filter on a half-sized texture and then double its size with bilinear filtering? Yeap… it would be a good approximation! Take a look at the results.



The images on the left present the results of applying a real 41×41 kernel size and the approximation. The approximation was done by downscaling the original 128×128 texture to 64×64, then to 32×32 and finally to 16×16. Then that tiny texture was blurred with a 5×5 kernel and then rescaled to 128×128 using bilinear filtering. The results look very good and the approximation is very good.

So we got the results of applying a 41×41 kernel on a 128×128 texture (which is not doable with today’s hardware) by applying a 5×5 kernel on a 16×16 texture (that is on 256 pixels!)… **very fast**. The 21×21 filter is approximated by applying the 5×5 kernel on the 32×32 texture, the 11×11 filter by applying the 5×5 kernel on the 64×64 texture and the 5×5 by applying the 5×5 kernel on the 128×128 texture. This way we get the 4 blurred textures that we add to the screen.

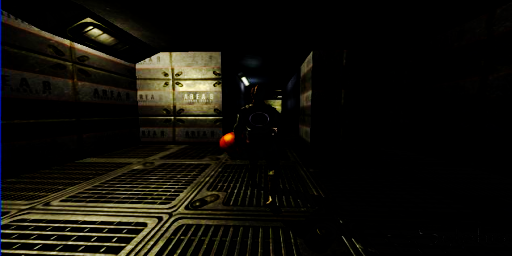
Lets look at how this look on a real scene…

**Example**

Lets see an example of how the two methods perform on a real scene. Below is a tone-mapped frame rendered from the [Sylphis3d](http://www.sylphis3d.com/) game engine:



The brightpass filter on that frame results in the next image:



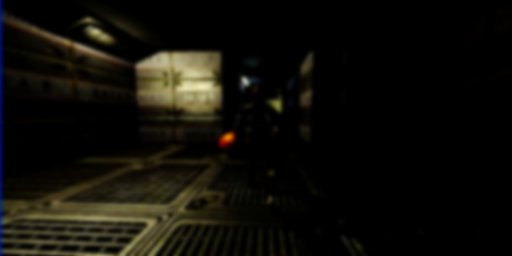
The above image consists of the parts that will glow. Applying a the conventional blur filter 4 times results in the image below:



Finally we add the blurred image on the original rendered frame and we get this result:



Even if the above image gives a nice bloom effect, the glare is minimal, and does not give a powerful effect. So lets see how the proposed blurring filter works on the same image. Beginning with the same brightpass image we blur by the *approximate* Gaussian filters that was described earlier in the post. We get these images for virtual kernel sizes of 5×5, 11×11, 21×21 and 41×41:

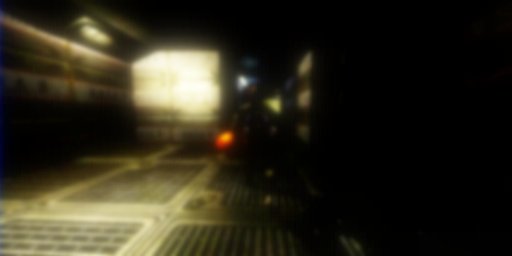




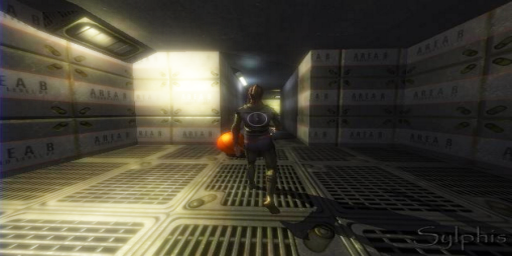




So adding up all the above images we get the final blurred brightpass image:



Adding this image on the original rendering we get this result:



I really think that the result speaks for itself! The blooming really got impressive and powerful. The fun thing is that we got this result by using less GPU horse power since we are blurring smaller textures!

I would also like to point out that this method gives very good results for LDR rendering, too. This is due to the fact that you can get wide bloom even from a constrained 0.0 to 1.0 range of colors, that would otherwise fade easily.

Hope you will find this useful and if you need any specific implementation details just ask!!

**If you liked this article [DIGG IT!!!](http://digg.com/programming/How_to_do_good_bloom_for_HDR_rendering)**  
**If you liked this article [BUMP IT!!!](http://www.devbump.com/story.php?title=How-to-do-good-bloom-HDR-rendering)**

hdr, opengl, direct3d, bloom, post process, effects, 3d, graphics, sylphis3d

(Visited 11,753 times, 14 visits today)

<http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/>