

## Screen Space Motion Blur on the GPU using CUDA

Katrin Schmid

### What is the problem you are trying to solve with this application?

Motion blur adds to the „realism“ of animation by enhancing the illusion of speed but is a costly effect to calculate. This project attempts to improve and speed up Image space motion blur using CUDA. Image space motion blur is a short cut to apply motion blur as a post process to an image of the scene based on object velocities.

### Describe your data in detail: where did it come from, how did you acquire it, what does it mean...

The model I am using was provided in CS175 class. This should also be tested with a bigger data set as I suspect caching of the graphics card is happening with smaller models and gives changed results. After a few runs there seems to be a major speedup in both the GLSL and CUDA version.

### Describe your program design and why you chose the features you did.

I am using OpenGL and GLSL to create the rendering and velocity data and vbo objects for displaying the geometry. CUDA is then used for blurring the image by the screen space vector.

The basic steps are:

1. render the scene to the frame buffer
2. create a velocity map (x, y velocity in screen space, length in z channel)
3. map the float velocity and colour texture so that its memory is accessible from CUDA
4. run CUDA to blur the image based on velocity vectors
5. copy result back
6. display the texture with a full screen quad



*Render only result, velocity map (GLSL), blurred image*

The model loader uses GLTools library (included in source). I compiled a version of the project on /home/kschmid/project/motionblur but seem to not be able to link it against glew (v.1.5.5) correctly on resonance.

### How do you use your application (mouse and keyboard functions, input/output, etc)?

A right mouse menu in the window shows all the user settable options which are:

- Enable CUDA or GLSL [space bar]
- animate [a]
- Toggle cube/ship [c]
- Render only [r]
- Velocity only [v]
- Decrease number of samples [n]
- Increase number of samples [m]
- Increase blur scale [+]
- Decrease blur scale [-]

- Reset timer [t]

In order for the ship model to be loaded correctly it must be in the same directory as the executable program while the .mtl must be in the source dir (we already noticed some bugs in CS175 but I didn't want to spend time on this).

**What is the performance of your code? What speedup and efficiency did you achieve? What optimizations did you implement to achieve this speedup?**

While speedup with traditional cpu based motion blurring should be obvious GLSL is already really fast. It's a quite narrow race but on my machine GLSL clearly wins out.

Below are the average frame rates on Windows32 with a Geforce 9600MGT card:

<u>Program</u>	<u>frame rate (fps)</u>		<u>average</u>	<u>speed comparison</u>
Run (1 minute averaged each):	1	2		
Render only	570			
Velocity map	520			
GLSL (16 samples)	293	284	288.5	0.92
CUDA (16 samples)	266	266	266	
GLSL (32 samples)	262	273	267.5	0.96
CUDA (32 samples)	267	249	258	
GLSL (64 samples)	300	279	289.5	0.88
CUDA (64 samples)	248	267	257.5	
GLSL (128 samples)	311	318	314.5	0.74
CUDA (128 samples)	202	264	233	

Overall the run times seem to vary a lot between different runs and during the run.

I am using fast math functions and constant memory for precalculated sample weights. I tried using shared memory but the maximum block size is only 23 pixel with float values which proved to small for this case. It might be well worth trying with integer as the values are in screen space and the precision loss might be acceptable.

While adaptive sampling (less samples for smaller velocities) seems a reasonable idea there was no speedup using it inside CUDA. A cut-off for small velocity values however proved valuable. Also splitting velocity to 2 position textures and halving velocity texture size didn't help much in improving speed. The best occupancy I could get is according to Visual profiler around 0.667

A detailed timing of the kernel code shows copying data back from CUDA takes about 50% of the total time of the kernel:

Preparation: 0.09592  
 Run kernel: 0.06167  
Copy data back: 0.22566  
**CUDA total: 0.38510**

I should test if this gets faster and the precision is still good enough when using 8 bit textures instead of float textures, same with shared memory. As the gap widens with more samples it is worth looking at the kernel again.

**What interesting insights did you gain from this project?**

I have been using this method of creating this method of creating motion blur for quite a while without thinking about the details. While speedup with traditional cpu based motion blurring should be obvious GLSL is already really fast. CUDA on the other hand is more flexible and gives the programmer much more control regarding memory etc... It would be also worth looking at how GLSL build-in interpolation in vertex shaders can be achieved in CUDA.

**What extensions and improvements can you suggest?**

In terms of speed the most promising future approach is to create velocity information yourself by passing the Vbo to CUDA and so avoid the additional step of having to create the velocity map in GLSL. This would also be able to handle deformable objects. The hard part is probably interpolation between vertices which GLSL does.

Pixel space motion blur has some limitations such as not supporting transparency, nondeformable objects and occlusion well and the linear blur it creates is unsuitable for some movements. All of this can be tackled to improve the quality.

**What did you most enjoy about working on this project? What was the most challenging aspect? What was the most frustrating? What would you do differently next time?**

The best experience about this project for me was understanding how OpenGL and CUDA play together. Speed optimization however turned out much harder than expected.

**Sources:****Code:**

- Ship loader and model from CS175 class
- Cg motion blur code demo code from NVIDIA (i translated to GLSL)
- CUDA Sdk examples: PostProcessGL and SimpleGL

**Papers:**

- Stupid OpenGL Shader Tricks Simon Green, NVIDIA
- Pixmotor: A Pixel Motion Integrator Ivan Neulander Rhythm & Hues Studios
- Pixel Motion Blur <http://ozlael.egloos.com> <http://www.slideshare.net/ozlael/motionblur-3252650>