Particle based fluid simulation on GPU

Aleš Koblížek
koblial2@fel.cvut.cz

Final report for B4M39GPU course
winter term 2019/2020

***Abstract*—There are two basic approaches to simulation of fluid dynamics: Eulerian – focusing on motion at specific location and Lagrangian – focusing on motion of individual fluid parcels. This paper takes the latter approach. The SPH (Smoothed particle hydrodynamics) computation method has been implemented in compute shader along with OpenGL visualization of the particles. A regular grid is used to accelerate the search for neighbour particles.**

I. Introduction

Fluids are very common in various real-world environments, therefore they are necessary for their virtual models to remain realistic. For that, however, a large number of particles is required to provide sufficient detail. Since there are no dependencies between the particles, their state can be updated independently in every step, which makes this task perfect for the massively parallel power of GPUs.

II. Smoothed particle hydrodynamics

Smoothed particle hydrodynamics is an approach introduced in [4] by Monaghan. Each particle i has constant mass m_i and rest density ρ_{0i} assigned according to the type of fluid. The particles form a density field

$$\rho(\mathbf{r}) = \sum_i m_i W_d(\mathbf{r} - \mathbf{r}_i, h) \tag{1}$$

which does not have to be bounded. W_d is a density kernel, it will be explained in the next section along with the other kernels. This density field generates forces $\mathbf{f}_i^{pressure}$, $\mathbf{f}_i^{viscosity}$ acting on the particles.

The simulation loop first computes the values of the density field at the location of each particle. These are then used to calculate the force acting on a particle

$$\mathbf{f}_i = \mathbf{f}_i^{pressure} + \mathbf{f}_i^{viscosity} + \mathbf{f}_i^{external}, \tag{2}$$

where

$$\mathbf{f}_i^{viscosity} = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho(\mathbf{r}_j)} W_v(\mathbf{r}_i - \mathbf{r}_j, h) \tag{3}$$

$$\mathbf{f}_i^{pressure} = -\sum_j m_j \frac{p(\mathbf{r}_i) + p(\mathbf{r}_j)}{2\rho(\mathbf{r}_j)} \mathbf{W}_p(\mathbf{r}_i - \mathbf{r}_j, h) \tag{4}$$

and $\mathbf{f}_i^{external}$ may be collision forces from other objects, gravity, or forces caused by user interaction. μ is the viscosity coefficient and

$$p(\mathbf{r}_i) = k(\rho(\mathbf{r}_i) - \rho_{0i}), \tag{5}$$

is the pressure at the location of the particle. k is the pressure coefficient. The particle position is then updated from its velocity

$$\mathbf{r}_i = \mathbf{r}_i + s\mathbf{v}_i, \tag{6}$$

s being the simulation step. The last step is to update the velocity from the previously computed forces

$$\mathbf{v}_i = \mathbf{v}_i + s\mathbf{a}_i, \tag{7}$$

$\mathbf{a}_i$ is acceleration of the particle obtained as

$$\mathbf{a}_i = \frac{\mathbf{f}_i}{\rho(\mathbf{r}_i)}. \tag{8}$$

Morris proposed in [5] to add a surface tension force, which should help to maintain sharp interfaces between fluids of different rest densities. For an overview of SPH with broader context, such as different neighbour search methods or integration methods, see [1].

A. Smoothing kernels

Müller proposed in [6] the following kernel to be used for computation of density

$$W_d(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - |\mathbf{r}|^2)^3 & 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{9}$$

Its advantage is that there is no square root in the distance computation. The next kernel is used for presure computation

$$\mathbf{W}_p(\mathbf{r}, h) = \frac{\mathbf{r}}{|\mathbf{r}|} \frac{45}{\pi h^6} (h - |\mathbf{r}|)^2 \tag{10}$$

and the last kernel – for computation of viscosity

$$W_v(\mathbf{r},h) = \frac{45}{\pi h^6}(h - |\mathbf{r}|) \qquad (11)$$

For desired properties of the kernels and a discussion of the advantages and disadvantages of those listed here, see [5].

III. Neighbour search

The time complexity of the SPH algorithm as described above is $O(n^2)$. But a particle is going to be affected only by those particles that are very close to it. Therefore it is possible to save time when computing the density and forces by including only the particles which are near enough. These can be found faster by building an auxilliary data structure before each update step of the particles.

First all the particles are wrapped in a bounding cube, which is partitioned into cells by a uniform square grid. Then a list of particles for each cell is created. The following two methods described by Green [2] can be used on a GPU. This paper uses the second one.

A. Atomic operations

This approach uses for each cell a counter that stores the number of particles and a fixed-size list of particle indices belonging to that cell. Updating the data structure is done with one thread per particle. The thread increments the counter and adds the particle index to the list. This causes scattered writes (and later scattered reads in the particle update phase) and potential bank conflicts.

B. Sorting

First a list a particle records needs to be built. Each record contains the $cellID$ of the cell it belongs to and the $particleID$. These records are then sorted by $cellID$. Then a list of cell records is built, each cell record contains ID of the first particle and number of particles in that cell. See figure 1 and 2. Finally, the particle data (position, velocity, etc.) is reordered according to the order of particle records. In the update phase, a particle is affected only by particles from the same cell and the cells surrounding it (27 cells in total = 3^3).

The size of the cell should be at most h, which is the radius of the kernel, because the weight for any particle that is farther away than h will be 0. The $cellID$ can be calculated from 3D coordinates of the particle by first calculating the cell coordinates as

$$\mathbf{cell}(\mathbf{r}) = s\frac{\mathbf{r} - \mathbf{BB}_{min}}{\mathbf{BB}_{max} - \mathbf{BB}_{min}}, \qquad (12)$$

where $\mathbf{BB}_{min}$ and $\mathbf{BB}_{max}$ are the bounding cube min and max coordinates respectively. The cell index is then calculated from cell coordinates as

$$cellID(\mathbf{c}) = c_1 s^2 + c_2 s + c_3. \qquad (13)$$

IV. Parallelization

At the beginning of every frame, the auxilliary data structure must be built. This consists of several steps, which need to be done one after another – without overlap.

1) Create particle records. (parallel)
2) Sort particle records. (parallel – using bitonic sort – see below)
3) Create cell records. (single thread on GPU)
4) Reorder particle positions and velocities. (parallel – different source and destination buffers)

Then the particle positions and velocities can be updated in the following non-overlapping steps.

5) Compute density at position of each particle. (parallel)
6) Update particle positions and velocities. (parallel)

A. Bitonic mergesort

Bitonic sort is a parallel sorting algorithm based on merging two sorted sequences (a bitonic sequence) into a single one. A bitonic sequence is a sequence that can be split into two sorted sequences, or its circular shift. Figure 3 shows a bitonic sorting network. The comparisons and swaps in single column of the red boxes can be executed in parallel. Afterwards, all threads must be synchronized. Therefore this column will correspond to a single compute shader dispatch. Listing 1 demonstrates this. Iteration of the inner for loop corresponds to the red-box column, iteration of the outer for loop corresponds to the blue/green-box column in the figure.

```
for(size_t b=2; b<2*ParticleN; b*=2) {
 for(size_t seqLen=b; seqLen>1; seqLen/=2) {
  glDispatchCompute(ParticleN/tGroupSize,1,1);
  glMemoryBarrier(...);
 }
}
```

Listing 1. Calls to the bitonic sort kernel

V. Implementation details – compute shaders

The sequence of kernels is described in section IV. It is necessary to ensure memory coherenecy between dispatches by using a memoryBarrier. All the steps which are marked "(parallel)" use the same number of

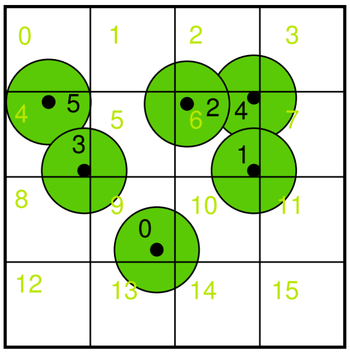

Figure 1. Particles in a grid. The green circle marks area with potential neighbours affecting the particle. From [2].

	Particle records unsorted $(cellID, particleID)$	Particle records sorted by $cellID$	cell records $(firstParticle, N)$
0	(9,0)	(4,3)	(-,0)
1	(6,1)	(4,5)	(-,0)
2	(6,2)	(6,1)	(-,0)
3	(4,3)	(6,2)	(-,0)
4	(6,4)	(6,4)	(0,2)
5	(4,5)	(9,0)	(-,0)
6			(2,3)
7			…

Figure 2. Creating the cell records by sorting the particle records.

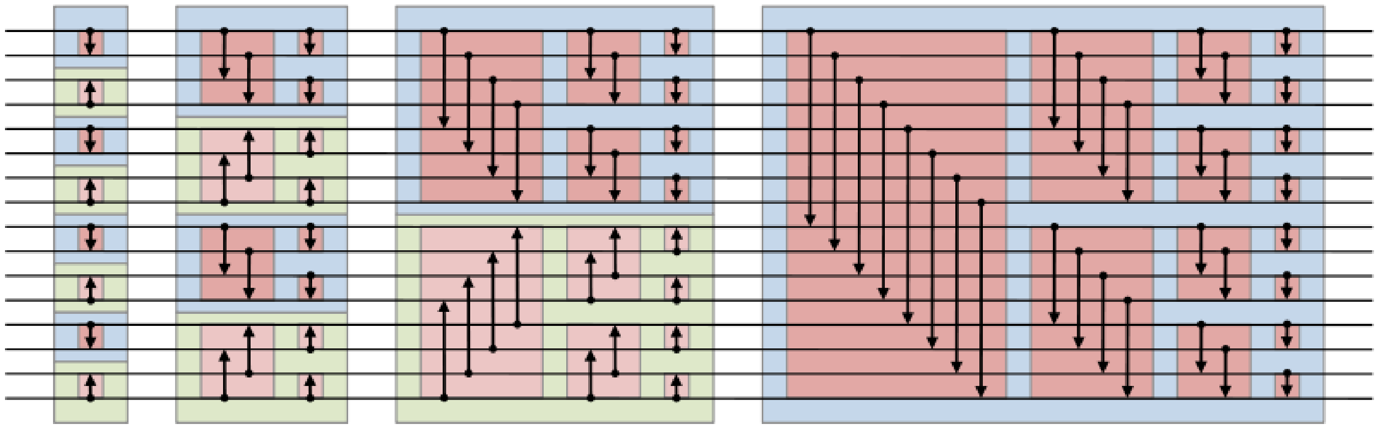

Figure 3. Bitonic sorting net. The elements to be sorted enter on the wires from the left and exit on the right in a sorted sequence. Arrows represent a comparison between two elements and conditional swap so that the arrow always points to the larger element. On the input of a blue or green block is always a bitonic sequence and on the output is a sorted sequence. Therefore, since the blue and green boxes have opposite sorting direction, the output of neighbouring blue and green box forms a bitonic sequence, which becomes an input for the next box. From Wikipedia.

threads (shader invocations) as there are particles. Sorting requires $log^2(particleN)$ iterations, same number of threads. Creating cell records needs single iteration over the sorted cell records of one thread. Shared memory is currently not used, but the steps 5) and 6) could be changed so they could potentially benefit from using it. See section IX for possible improvements.

VI. User interface

The application can be controlled by keys listed in table I. Simulation parameters can be increased (multiplied by two) with upper case letters, decreased (divided by two) by lower case letters as listed in the table. After adjusting a parameter, the current setting of all parameters is printed to standard output.

VII. Parameter choice and testing

The SPH method has several parameters. The currently used values listed in table II have been determined empirically. The application was tested visually with these parameters for varying number of particles and NN grid resolution. The fluid behaves strangely for large number of particles, as if it was compressing and expand-

Key	**Action**
esc	exit
r	restart simulation
Key (+/-)	**Parameter**
S/s	simulation step
H/h	kernel radius
M/m	particle mass
K/k	pressure coefficient
U/u	viscosity coefficient

Table I
Application controls

ing, possibly due to suboptimal choice of coefficients. Particles occasionally get stuck on the boundary. Also because the step size is constant and relatively large, the particles never stop moving. Figure 4 shows a screenshot of the application.

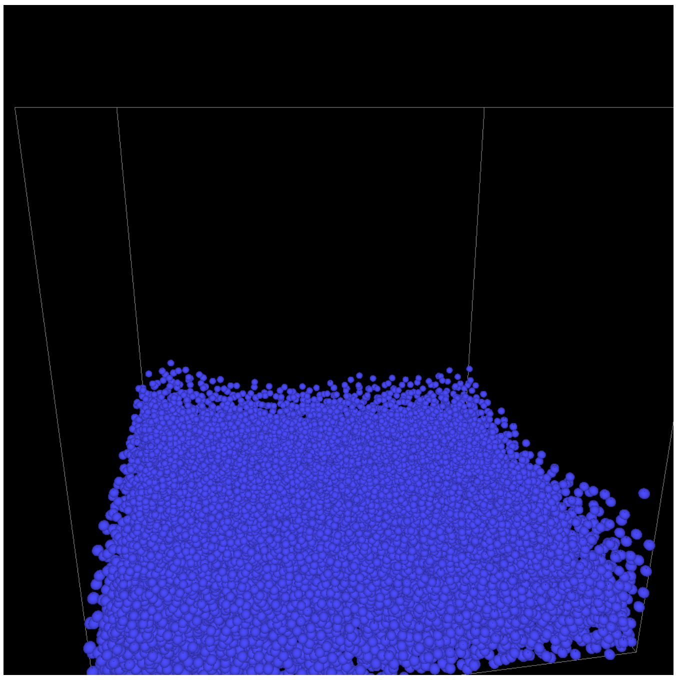

Figure 4. Screenshot from the application for 16384 particles.

Parameter	Value
Simulation step	0.005 seconds
Kernel radius	0.1
Particle mass	32
Pressure coefficient	2.4
Viscosity coefficient	2048

Table II
CURRENT SIMULATION PARAMETERS.

VIII. Performance

The measured results in table V were obtained on the configurations listed in table III and IV. They are dependent on number of particles in each cell, which in turn depends on the pressure and viscosity coefficients.

As expected, the frame time grows more or less linearly with the number of particles. There is an unexpected increase in frame time between the 5. and 6. row due to increased number of cells, which shoud be investigated. Also when comparing the results with other implementations, there seems to be higher penalty for increasing the number of cells, but it handles better higher numbers of particles.

PC-1
Debian 11 GNU/Linux 4.19.0-6-amd64
Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz
6 GB RAM
NVIDIA GeForce GT 750M (GK107M)
OpenGL Driver version 4.6.0
NVIDIA proprietary driver version 418.74
384 CUDA cores, 2 GB VRAM

Table III
PC-1 USED TO MEASURE THE RESULTS.

PC-2
MS Windows 10 64-bit
Intel Xeon E3-1231 v3 @ 3.4 GHz
16 GB RAM
NVIDIA GeForce GTX 1070 Ti
NVIDIA driver version 432.00
2432 CUDA cores, 8 GB VRAM

Table IV
PC-2 USED TO MEASURE THE RESULTS.

IX. Conclusion

Fluid simulation using the SPH method has been implemented on GPU. The demo application has minor bugs, such as particles getting stuck on boundary. Also the empirically determined parameters may not be the best choice – there are ways to determine them described for example in [3]. Currently the euler integration method is used, but the leapfrog integration method is said to be more stable. The performance is much higher than on the CPU, but the implementation should still be improved if it was inteded for real-time application running 60 frames per second, which gives only about 17 ms for the entire frame. There seems to be unnecessarily higher penalty for higher number of cells, which should be investigated. On the other hand, it seems to handle higher number of particles better.

The GPU implementation could most likely be im-

	Grid subdivision (s) (along each dimension)	**# of particles**	**PC-1**	**PC-2**
1	16	32 768	47	8
2	32	32 768	23	6
3	32	65 536	69	12
4	64	32 768	38	12
5	64	262 144	220	54
6	64	524 288	437	103
7	128	524 288	477	152
8	128	1 048 576	835	250
9	128	2 097 152	1 622	445

Table V
MEASURED AVERAGE SIMULATION FRAME TIME (WITHOUT VISUALIZATION) IN MILLISECONDS.

proved. Firstly, the shader that creates cell records could be changed to use one thread per particle. Each thread would compare the ID of the cell it belongs to with that of the previous particle. If it is different, it indicates the beginning of the new cell in the sorted sequence of particle records and the thread would write its ID to the cell record. Second pass would work the same way but it would look for the end of the sequence of particle records and, using the beginning obtained in the previous pass, it would write the length of the sequence into the cell record.

Another possible improvement could be made in the density computation and particle update shaders. Each cell record would be processed by one workgroup, therefore particles in the surrounding cells could be loaded into shared memory and could be used for updating all particles in this cell.

The implementation of bitonic sort could also be changed so that when the height of the red block in figure 3 becomes smaller than double of the number of threads per block, it is then sufficient to synchronize only threads within each block and the shared memory can also be used.

References

[1] Ertekin, B.: *Fluid Simulation using Smoothed Particle Hydrodynamics*. 2015.
[2] Green, S.: *Particle Simulation using CUDA*. 2010.
[3] Mao et al.: *A comprehensive study on the parameters setting in smoothed particle hydrodynamics (SPH) method applied to hydrodynamics problems*. 2017.
[4] Monaghan, J.J.: *Smoothed Particle Hydrodynamics*. 1992.
[5] Morris, J.P.: *Simulating surface tension with smoothed particle hydrodynamics*. 2000.
[6] Müller et al.: *Particle-Based Fluid Simulation for Interactive Applications*. 2003.