Real-Time GPU-based SPH Fluid Simulation Using Vulkan and OpenGL Compute Shaders

Samuel I. Gunadi, Pujianto Yugopuspito
Informatics Dept., Universitas Pelita Harapan
samuel.i.gunadi@gmail.com, p.yugopuspito@ieee.org

***Abstract*—We investigate the potential of Vulkan for fluid animation using smoothed particle hydrodynamics (SPH) method. A simple parallel SPH algorithm is devised and implemented in OpenGL Shading Language(GLSL) compute shader. This GLSL code is converted into latest Standard Portable Intermediate Representation (SPIR-V) format and then implemented with Vulkan and OpenGL 4.6. Two test cases are used to verify that the implementations are correct by examining the visual output. The first test case is dropping a cube of water into a box and the second is a dam break in a closed channel, both at room temperature. And finally, we analyze the performance of both implementations in real-time graphic processor unit (GPU). We found that the most optimum of the work group size is 128, and our Vulkan implementation performs faster than OpenGL for 30,000 or greater particles.**

***Index Terms*—hydrodynamics , OpenGL, Vulkan, simulation, performance, shader, real-time GPU.**

I. Introduction

During the past few years, the increase of computational power has been realized using more processors with multiple cores and specific processing units like graphics processing units (GPUs). The computational power of graphical processing units (GPUs) on modern video cards often surpasses the computational power of the CPU that drives them. Unlike CPU that works at really high frequency to achieve high speed, GPUs have a parallel architecture composed of many streaming multiprocessors that work at a lower frequency allowing for lower power consumption and faster execution time if the algorithm is parallelizable (able to be made parallel) [1]. The performance of computation by means of compute time can be improved [2].

Though Eulerian approach was a popular scheme for fluid animation, it has a few important drawbacks such as: it needs global pressure correction and has poor scalability. Due to these drawbacks, such schemes are unable to take benefits of parallel architectures available today. Particle-based methods are free from these limitations and hence are becoming more popular in fluid animation.

The smoothed particle hydrodynamics (SPH) is a mesh-free, Lagrangian, particle method for modelling fluid flow. This method was first introduced by Gingold and Monaghan in 1977 [3] originally for modelling astrophysics phenomena and later extended for modeling fluid phenomena. It works by partitioning the fluid volume into discrete particles, and a single particle represents a small fraction of the volume. It is *smoothed* because it blurs out the boundaries of a particle so that a smooth distribution of physical quantities is obtained [4]. In [5], SPH is used to represent the simulated atherosclerotic blood vessel.

The first implementation of the SPH method totally on GPU was realized by Harada et al. [6] in 2007 using OpenGL and Cg (C for graphics). Harada demonstrated 60 000 particles fluid animation at 17 frames per second which is much faster compared to CPU based SPH fluid animation. Since then, there has been growing interest in the implementation of SPH on the GPU resulting in several implementations that take full or partial advantage of the GPU. Harada et al. used some tricks to work around the restrictions imposed by the languages. In particular, since OpenGL did not offer the possibility to write to three-dimensional textures, the position and velocity textures had to be flattened to a two-dimensional structure.

Vulkan is a new graphics and compute application programming interface (API) by Khronos Group. As SPH implementation s not available yet in Vulkan, this paper reports an experience of expanding basic Vulkan to represent the SPH fluid simulation in realtime, and investigate its potential for fluid animation. Furthermore we compare it with the established OpenGL by means as a compute shader.

II. Modelling Fluids

The most well-known model for describing the behavior of fluids is given by the Navier–Stokes equations. These equations model a fluid by considering the physical quantities mass-density, pressure and velocity as continuous fields and describing their relationships over time with differential equations.

A. Navier–Stokes Equation

The Navier–Stokes equations for incompressible, isothermal Newtonian fluids with constant viscosity [7], [8] in vector notation are the momentum equation

$$\rho(\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v}) = -\nabla p + \rho \vec{g} + \mu \nabla^2 \vec{v} \tag{1}$$

or

$$\rho \frac{D\vec{v}}{Dt} = -\nabla p + \rho \vec{g} + \mu \nabla^2 \vec{v} \tag{2}$$

and the continuity equation

$$\nabla \cdot \vec{v} = 0. \tag{3}$$

Newtonian fluids are defined as fluids for which the shear stress is linearly proportional to the shear strain rate [9].

Equation 1 has been arranged so that the acceleration terms are on the left side and the force terms are on the right side, where ρ is the density, $\vec{v}$ is the flow velocity, p is the pressure, $\vec{g}$ is an external force density field (including gravity), and μ is the viscosity of the fluid.

B. Smoothed Particle Hydrodynamics

The key idea behind smoothed particle hydrodynamics is the integral approximation of a field function [10]. The integral representation of a field function A is

$$A(r) = \int_{\omega} A(r')W(r - r', h)dr' \tag{4}$$

where r is a point in space, ω is the volume that contains r, and W is a kernel function with a smoothing radius h. If W is the Dirac delta function, the integral representation reduces to the exact value of $A(r)$, otherwise it is known as a kernel approximation.

The integral representation can be discretized by a particle approximation, which is the summation of all particles

$$A(\vec{r}) = \sum_j A_j V_j W(\vec{r} - \vec{r}_j, h), \tag{5}$$

where j iterates over all particles, A_j is the field value at coordinate r_j, and V_j is the volume of particle j. For a fluid, the volume of a particle is its mass divided by its density

$$V = \frac{m}{\rho}. \tag{6}$$

Thus the particle approximation of a field function in SPH is

$$A(\vec{r}) = \sum_j A_j \frac{m_j}{\rho_j} W(\vec{r} - \vec{r}_j, h), \tag{7}$$

C. Computing Density

Applying the SPH approximations to the Lagrangian formulation of the Navier–Stokes equations is done in two steps. By keeping the mass fixed for each particle, the continuity equation can be omitted, and the density of each particle is approximated with Equation 7

$$A(\vec{r}) = \sum_j \rho_j \frac{m_j}{\rho_j} W(\vec{r} - \vec{r}_j, h), \tag{8}$$

which is then simplified to

$$\rho_i = \sum_j m_j W(\vec{r}_i - \vec{r}_j, h). \tag{9}$$

D. Computing Pressure

The pressure P_i of particle i is approximated with the equation of state to give weak compressibility, rather than solve the Poisson pressure equation for near incompressibility, to reduce computational cost

$$P = R\rho T, \tag{10}$$

where P is pressure, R is the constant value for each gas, ρ is density and T is temperature. The equation can also be written as

$$P = k\rho, \tag{11}$$

where k is a gas stiffness constant that depends on the temperature. As suggested by Desbrun [11], the equation is then modified to

$$P_i = k(\rho_i - \rho_0), \tag{12}$$

where ρ_i is the density of particle i and ρ_0 is the resting density.

E. Computing Acceleration

The acceleration of a fluid particle can be expressed as

$$\vec{a} = \frac{D\vec{v}}{Dt} = \frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} = \frac{\partial \vec{v}}{\partial t} + u\frac{\partial \vec{v}}{\partial x} + v\frac{\partial \vec{v}}{\partial y} + w\frac{\partial \vec{v}}{\partial z}, \tag{13}$$

but since the particles move with the fluid, the convective term $\vec{v} \cdot \nabla \vec{v}$ is not needed

$$\vec{a}_i = \frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{\rho_i}, \tag{14}$$

where $\vec{v}_i$ is the velocity of particle i, $\vec{F}_i$ is the force density field of particle i and ρ_i is the density field evaluated at the location of particle i.

F. Computing Forces

The momentum equation (Equation 1) can be separated into three components on the right hand side

$$\frac{d\vec{v}_i}{dt} = \vec{F}_i^{\text{pressure}} + \vec{F}_i^{\text{viscosity}} + \vec{F}_i^{\text{external}}. \tag{15}$$

The pressure force F_i^{pressure} can be computed by applying Equation 7 to the pressure term $-\nabla p$

$$\vec{F}_i^{\text{pressure}} = -\nabla p(\vec{r}_i) = -\sum_j m_j \frac{P_j}{\rho_j} \nabla W(\vec{r}_i - \vec{r}_j, h). \tag{16}$$

However this does not produce a symmetric force and violates the action-reaction law (action is not equal to reaction), as can be seen when only two particles interact. Since the gradient of the kernel is zero at its center, particle i only uses the pressure of particle j to compute its pressure force and vice versa. Because the pressures at the locations of the two particles are not equal in general, the pressure forces will not be symmetric. So a symmetric approximation of the gradient is used instead [12]

$$\vec{F}_i^{\text{pressure}} = -\sum_j m_j \frac{P_i + P_j}{2\rho_j} \nabla W(\vec{r}_i - \vec{r}_j, h). \tag{17}$$

The viscosity force $F_i^{\text{viscosity}}$ can be computed by applying Equation 7 to the viscosity term $\mu \nabla^2 v$

$$\vec{F}_i^{\text{viscosity}} = \mu \nabla^2 \vec{v}(\vec{r}_i) = \mu \sum_j m_j \frac{\vec{v}_j}{\rho_j} \nabla^2 W(\vec{r}_i - \vec{r}_j, h). \tag{18}$$

Again this is not symmetric. To make it symmetric, velocity differences are used [12]

$$\vec{F}_i^{\text{viscosity}} = \mu \sum_j m_j \frac{\vec{v}_j - \vec{v}_i}{\rho_j} \nabla^2 W(\vec{r}_i - \vec{r}_j, h). \tag{19}$$

G. Leapfrog Integration

The advantage of the leapfrog integration is its low memory requirement and the efficiency for one force evaluation per step. The value of the time step is important for the stability of the simulation. Setting the time step too high can result in erratic simulations.

With constant time step dt, the new particle position $\vec{r_i}'$ and velocity $\vec{v_i}'$ at time t_{n+1} is computed with

$$\vec{v_i}' = \vec{v_i} + dt\frac{\vec{F}}{\rho_i} \tag{20}$$

$$\vec{r_i}' = \vec{r_i} + dt\ \vec{v_i}. \tag{21}$$

H. Kernel Functions

The stability, accuracy, and speed of the SPH method depends heavily on the kernel functions. Each kernel function is designed for different purposes. Kernel functions and their derivatives must have compact support, having a value of 0 outside of the smoothing radius. The following smoothing kernels from [12] were selected for their performance.

1) Poly6 Kernel: The poly6 kernel is also known as the sixth-degree polynomial kernel. The poly6 kernel is used for density computation (Equation 9).

$$W_{\text{poly6}}(\vec{r}, h) = \frac{315}{64\pi h^9}\begin{cases}(h^2 - \|\vec{r}\|^2)^3, & 0 \le \|\vec{r}\|^2 \le h\\ 0, & \text{otherwise}\end{cases} \tag{22}$$

The gradient of this kernel is used for surface normal.

$$\nabla W_{\text{poly6}}(\vec{r}, h) = -\frac{945}{32\pi h^9}\begin{cases}\vec{r}(h^2 - \|\vec{r}\|^2)^2, & 0 \le \|\vec{r}\|^2 \le h\\ 0, & \text{otherwise}\end{cases} \tag{23}$$

The Laplacian of this kernel is used for computing surface tension.

$$\nabla^2 W_{\text{poly6}}(\vec{r}, h) = \\ -\frac{945}{32\pi h^9}\begin{cases}(h^2 - \|\vec{r}\|^2)(3h^2 - 7\|\vec{r}\|^2), & 0 \le \|\vec{r}\|^2 \le h\\ 0, & \text{otherwise}\end{cases} \tag{24}$$

2) Spiky Kernel: If poly6 kernel is used for the computation of pressure force, particles tend to build clusters under high pressure [12]. As particles get very close to each other, the repulsive force vanishes since the gradient of the kernel approaches zero at the center. To solve this problem, the spiky kernel proposed by Desbrun and Gascuel [11] is used

$$W_{\text{spiky}}(\vec{r}, h) = \frac{15}{\pi h^6}\begin{cases}(h - \|\vec{r}\|)^3, & 0 \le \|\vec{r}\| \le h\\ 0, & \text{otherwise}\end{cases} \tag{25}$$

The gradient of spiky kernel is used to compute the pressure force (Equation 17).

$$\nabla W_{\text{spiky}}(\vec{r}, h) = -\frac{45}{\pi h^6}\begin{cases}(h - \|\vec{r}\|)^2\hat{r}, & 0 \le \|\vec{r}\| \le h\\ 0, & \text{otherwise}\end{cases} \tag{26}$$

3) Viscosity Kernel: Viscosity is a property of fluid that comes from collisions between particles that moves at different velocities. However, for two particles that get close to each other, the Laplacian of the smoothed velocity field can get negative resulting in forces that increase their relative velocity [12]. Because of this, artifacts may appear in coarsely sampled velocity fields. To solve this problem, a kernel whose Laplacian is positive everywhere is used.

$$W_{\text{viscosity}}(\vec{r}, h) = \\ \frac{15}{2\pi h^3}\begin{cases}-\frac{\|\vec{r}\|^3}{2h^3} + \frac{\|\vec{r}\|^2}{h^2} + \frac{h}{2\|\vec{r}\|} - 1, & 0 \le \|\vec{r}\| \le h)\\ 0, & \text{otherwise}\end{cases} \tag{27}$$

The Laplacian of this kernel is used for computing viscosity force (Equation 19).

$$\nabla^2 W_{\text{viscosity}}(\vec{r}, h) = \frac{45}{\pi h^6}(h - \|\vec{r}\|) \tag{28}$$

I. Neighbor Search

In SPH, it is important to efficiently and quickly find the neighboring particles in a given radius. The nave approach is to search over all particles, but it is computationally expensive. Its computational complexity is $\mathcal{O}(n^2)$. The solution is to use a spatial subdivision data structure called a uniform grid whose cell size is the smoothing length h. This reduces the computational complexity to $\mathcal{O}(nm)$, where n is the number of cells and m is the average number of particles per grid cell [12]. However, we do not use a uniform grid in our implementation because of GLSL limitations.

The pseudocode of the simulation loop is shown in Algorithm 1.

III. Implementation Details

Real-time (greater than 60 frames per second) SPH simulation is implemented in Vulkan and OpenGL 4.6 (C++) and compute shaders (GLSL). The following is the details of how we implement it.

A. Data structures

The particle data structure contains all the attributes necessary for a particle: position, velocity, force, density, and pressure. We use a flat data structure whose elements are stored together in a contiguous piece of storage. Flat data structures have an advantage in cache locality that makes them more efficient to traverse [13]. We initially used `struct`, but the flat data structure has better cache locality which led to fewer cache misses. The performance gain is significant, especially with lower number of particles.

B. Compute shader

A compute shader is one of the shader stages and its purpose is to carry out computations. In both OpenGL and Vulkan, the compute pipeline is separate from the graphics pipeline. In our Vulkan implementation, a total of four pipelines are used, consisting of 3 compute pipelines and 1 graphics pipeline. Each compute pipeline contains one shader module.

Algorithm 1 Simulation loop of parallel SPH based on equation of state

```
1: for each particle $i$ in particles do in parallel ▷ Compute density and pressure
2:     $\rho_i \leftarrow 0$
3:     for each particle $j$ in particles do
4:         $\vec{r} \leftarrow \vec{r}_i - \vec{r}_j$
5:         if $\|\vec{r}\| < h$ then
6:             $\rho_i \leftarrow \rho_i + m_i \; W_{\text{poly6}}(\vec{r}, h)$
7:         end if
8:     end for
9:     $p_i \leftarrow k(\rho_i - \rho_0)$
10: end for
11: for each particle $i$ in particles do in parallel ▷ Compute forces
12:     $\vec{F}_i^{\text{pressure}} \leftarrow 0$
13:     $\vec{F}_i^{\text{viscosity}} \leftarrow 0$
14:     for each particle $j$ in particles do
15:         if $i \neq j$ then
16:             $\vec{r} \leftarrow \vec{r}_i - \vec{r}_j$
17:             if $\|\vec{r}\| < h$ then
18:                 $\vec{F}_i^{\text{pressure}} \leftarrow \vec{F}_{\text{pressure}} - m_i \; (p_i + p_j)/2\rho_j \; \nabla W_{\text{spiky}}(\vec{r}, h)$
19:                 $\vec{F}_i^{\text{viscosity}} \leftarrow \vec{F}_{\text{viscosity}} + m_i \; (\vec{v}_j - \vec{v}_i)/\rho_j \; \nabla^2 W_{\text{viscosity}}(\vec{r}, h)$
20:             end if
21:         end if
22:     end for
23:     $\vec{F}_i^{\text{viscosity}} \leftarrow \vec{F}_i^{\text{viscosity}} \; \mu$
24:     $\vec{F}_i^{\text{external}} \leftarrow \vec{F}^{\text{gravity}} \; \rho_i$
25:     $\vec{F}_i^{\text{total}} \leftarrow \vec{F}_i^{\text{pressure}} + \vec{F}_i^{\text{viscosity}} + \vec{F}_i^{\text{external}}$
26: end for
27: for each particle $i$ in particles do in parallel ▷ Leapfrog integration
28:     $\vec{a}_i \leftarrow \vec{F}_{\text{total}}/\rho_i$
29:     $\vec{v}_i \leftarrow \vec{v}_i + \Delta t \; \vec{a}_i$
30:     $\vec{r}_i \leftarrow \vec{r}_i + \Delta t \; \vec{v}_i$
31: end for
```

All GLSL source files are compiled into SPIR-V [14] first using Khronos Group's glslang. Note that both our Vulkan and OpenGL implementations share the same GLSL code, except the part where OpenGL y-axis is pointing up, while Vulkan y-axis is pointing down. Detail source code of the OpenGL and Vulkan complete code can be found in our Github website [15] and [16]

C. Data dependency and synchronization

Synchronization is the forced ordering of executions, which is important to prevent race conditions. Algorithm 1 itself has several data dependencies, so it is divided into 3 compute shaders:

- The first reads the position then writes the density and pressure in parallel. It uses data computed by the third compute shader.
- The second reads the position, velocity, pressure, and density then writes the force in parallel. It uses data computed by the first compute shader.
- And the third reads the force then writes the position and velocity in parallel. It uses data computed by the second compute shader.

Memory barrier and pipeline barrier are used as the synchronization method. They are placed between invocations of compute shaders and between compute and graphics pipeline. This ensures that previous shader has finished writing new data before reading it (read after write [RAW] dependency).

D. Work division and work groups

The work carried out by a compute shader is divided into work groups, each of which can execute a given number of threads. Work groups are executed one at a time, but the threads of the work group are executed in parallel. The SPH algorithm is parallelized by assigning a thread to each particle in the simulation. Each thread is then responsible for calculating the SPH sums over the surrounding particles.

Vulkan and OpenGL automatically assign the work groups to SMs (streaming multiprocessors). It is important that the correct number of work groups are used. Higher *occupancy* does not always mean higher performance, but low occupancy always results in degraded performance. The *occupancy* is a ratio of the number of threads that can run in parallel in the compute shader to the maximum number of threads the GPU supports.

The work group size is set to the optimal value obtained by performing tests using our 20,000-particle OpenGL implementation, Fig. 1. Based on our empirical data (Table 1), we then set the work group size to 128 in both our Vulkan and OpenGL implementations. Consequently, we set the work group count to 157, i.e., the ceiling of the particle number (20,000) divided by the work group size (128).

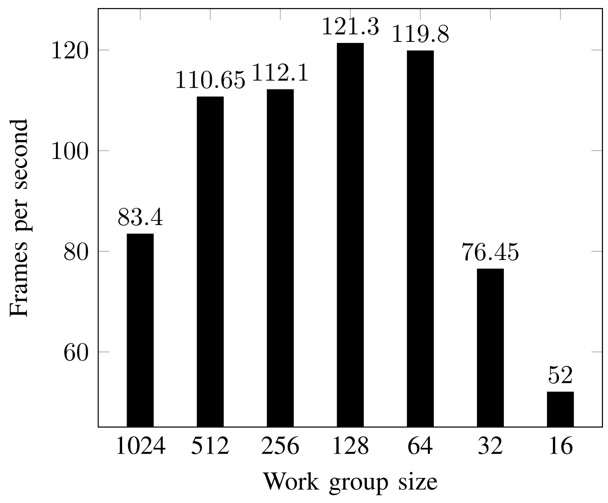

Fig. 1. Work group size to frame rates, 20,000-particle

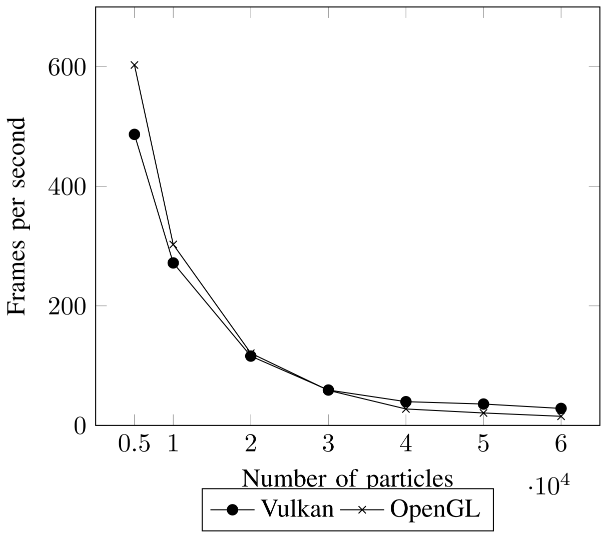

Fig. 2. Frame rates comparison

IV. Performance analysis

Our implementations were tested on a computer with GTX 1070 and i5-6600K at 3.50 GHz, and 16 GB RAM in dual channel. GTX 1070 has 1920 CUDA cores divided into 15 streaming multiprocessors, 98 304 B shared memory per multiprocessor, and 2048 max threads per multiprocessor. The operating system was Windows 10.0.16299.125 64-bit with Visual Studio 2017 15.3.3, Vulkan SDK 1.0.65.1, and NVIDIA video driver 390.77 installed. The third-party libraries used are: OpenGL Mathematics (GLM) 0.9.8.5, The OpenGL Extension Wrangler Library (GLEW) 2.1.0, and GLFW 3.2.1. The render result is shown in Figure 3 and Figure 4.

Figure 2 shows the implementations' frame rates with different number of particles. The frame rates were obtained from 20 seconds of runtime. Our Vulkan implementation is faster with higher number of particles, but performs worse with lower number of particles.

Our objective is to create real-time simulation, which means the computations must be fast enough that the results can be viewed immediately. Being able to conduct operations at 60 Hz or higher is considered real-time [17]. So the chosen number of particles is 20,000. The frame rate of our 20,000-particle Vulkan implementation is 115.95 fps, and 120.8 fps is the frame rate of its OpenGL 4.6 counterpart. This can be explained by the Vulkan characteristics: lower overhead and more direct control over the GPU and lower CPU usage [18]. Furthermore SPIR-V format is in fact a shader pre-compilation that allow application initialization speed is improved and a larger variety of shaders can be used per scene [14].

V. Conclusion

In this work, we investigate the potential of Vulkan for fluid animation using SPH method. We begin by devising a simple parallel SPH algorithm. Then we implement this algorithm in GLSL compute shader. Then we proceed to develop Vulkan and OpenGL implementations in GLSL compute shader in SPIR-V format. With 2 test cases, we verify that the implementations are correct by examining the visual output, and then measure its performance.

If the number of particles is 30,000 or greater, our Vulkan implementation performs faster compared to our OpenGL implementation. But our Vulkan implementation performs worse compared to our OpenGL implementation if the number of particles is 20,000 or fewer.

TABLE I
Frame rates table

Number of particles	Vulkan (fps)	OpenGL (fps)
5000	486.80	602.90
10000	271.80	302.65
20000	115.95	120.80
30000	59.15	58.65
40000	39.70	27.50
50000	35.75	20.80
60000	28.40	15.25

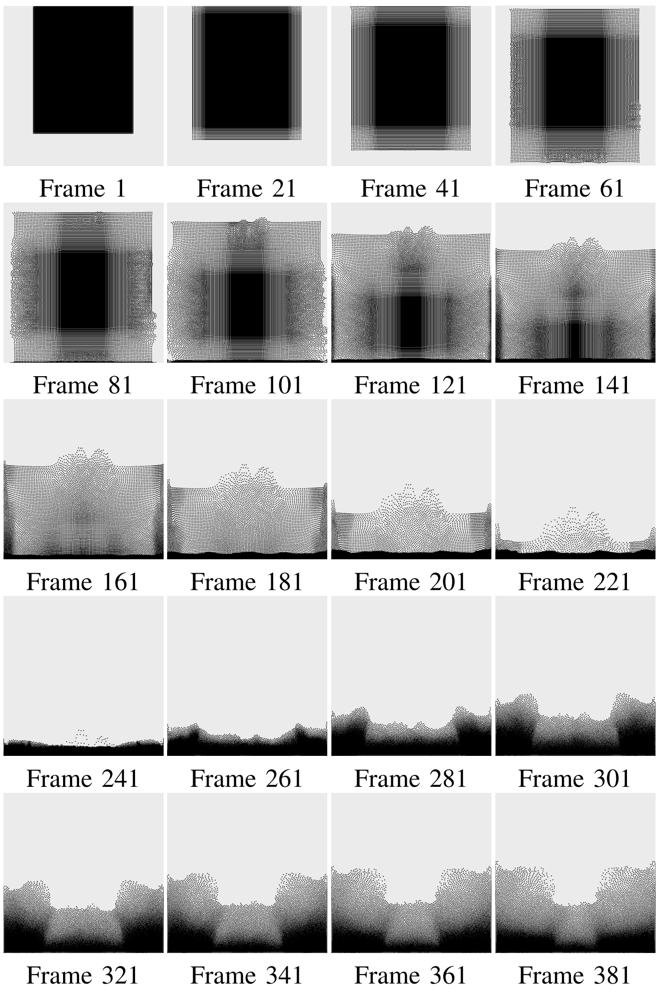

Fig. 3. Rendering of the first test case

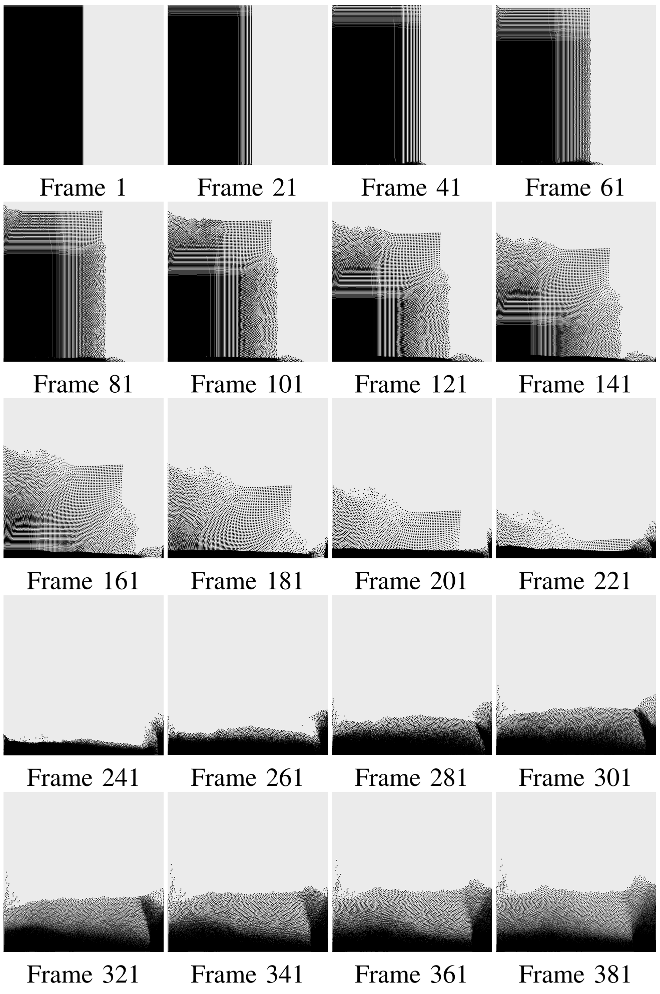

Fig. 4. Rendering of the second test case

REFERENCES

[1] Y. Cai and S. See, Eds., *GPU Computing and Applications*. Springer, 2015.

[2] P. Yugopuspito, Sutrisno, and R. Hudi, "Breaking through memory limitation in GPU parallel processing using strassen algorithm," in *2013 International Conference on Computer, Control, Informatics and Its Applications (IC3INA)*. IEEE, Nov 2013. [Online]. Available: https://doi.org/10.1109/ic3ina.2013.6819174

[3] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics: theory and application to non-spherical stars," *Monthly Notices of the Royal Astronomical Society*, vol. 181, no. 3, pp. 375–389, 1977. [Online]. Available: http://dx.doi.org/10.1093/mnras/181.3.375

[4] D. Kim, *Fluid Engine Development*. CRC Press, 2017.

[5] K. Wiratama, P. Yugopuspito, and H. Margaretha, "Performance evaluation of simulated smoothed particle hydrodynamics method in pulsating atherosclerotic blood vessel," in *2016 International Conference on Informatics and Computing (ICIC)*. IEEE, 2016. [Online]. Available: https://doi.org/10.1109/iac.2016.7905752

[6] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Smoothed particle hydrodynamics on gpus," in *Computer Graphics International*. SBC Petropolis, 2007, pp. 63–70.

[7] B. R. Munson, T. H. Okiishi, W. W. Huebsch, and A. P. Rothmayer, *Fundamentals of Fluid Mechanics*, 7th ed. Wiley, 2013.

[8] P. M. Gerhart, A. L. Gerhart, and J. I. Hochstein, *Munson, Young and Okiishi's Fundamentals of Fluid Mechanics*, 8th ed. Wiley, 2016.

[9] Y. A. Aengel and J. M. Cimbala, *Fluid Mechanics*, 3rd ed. McGraw-Hill Education, 2014.

[10] J. J. Monaghan, "Smoothed particle hydrodynamics," *Reports on Progress in Physics*, vol. 68, no. 8, p. 1703, 2005. [Online]. Available: http://stacks.iop.org/0034-4885/68/i=8/a=R01

[11] M. Desbrun and M.-P. Gascuel, *Smoothed Particles: A new paradigm for animating highly deformable bodies*. Vienna: Springer Vienna, 1996, pp. 61–76. [Online]. Available: https://doi.org/10.1007/978-3-7091-7486-9_5

[12] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '03. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 154–159. [Online]. Available: http://dl.acm.org/citation.cfm?id=846276.846298

[13] K. Guntheroth, *Optimized C++*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2016.

[14] J. Kessenich, "An introduction to spir-v: A khronos defined intermediate language for native representation of graphical shaders and compute kernels," 2015. [Online]. Available: https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf

[15] S. I. Gunadi, "Sph simulation in opengl compute shader." [Online]. Available: https://github.com/multiprecision/sph_opengl

[16] ——, "Sph simulation in vulkan compute shader." [Online]. Available: https://github.com/multiprecision/sph_vulkan

[17] S. Marschner and P. Shirley, *Fundamentals of Computer Graphics*, 4th ed. Natick, MA, USA: A. K. Peters, Ltd., 2016.

[18] A. Shilov, "Amd: Vulkan absorbed best and brightest parts of mantle," 2015. [Online]. Available: https://www.kitguru.net/components/graphic-cards/anton-shilov/amd-vulkan-absorbed-best-and-brightest-parts-of-mantle/