THESIS

REAL-TIME GPU-BASED SPH FLUID SIMULATION USING VULKAN AND OPENGL COMPUTE SHADERS

submitted in partial fulfillment of the requirements for the degree of

Sarjana Informatika Strata Satu

by

NAME	**: SAMUEL IVAN GUNADI**
STUDENT ID	**: 11220120018**

DEPARTMENT OF INFORMATICS
FACULTY OF COMPUTER SCIENCE
UNIVERSITAS PELITA HARAPAN
TANGERANG
2018

UNIVERSITAS PELITA HARAPAN
FAKULTAS ILMU KOMPUTER

PERNYATAAN KEASLIAN KARYA TUGAS AKHIR

Saya, sebagai mahasiswa Teknik Informatika di Universitas Pelita Harapan,

Nama	**: Samuel Ivan Gunadi**
NIM	**: 11220120018**
Program Studi	**: Teknik Informatika**
Konsentrasi	**: Rekayasa Piranti Lunak**

menyatakan bahwa karya tugas akhir yang saya buat dengan judul **REAL-TIME GPU-BASED SPH FLUID SIMULATION USING VULKAN AND OPENGL COMPUTE SHADERS**:

1. dibuat dan diselesaikan sendiri, dengan menggunakan hasil kuliah, tinjauan lapangan dan buku-buku serta jurnal acuan yang tertera di dalam referensi pada karya tugas akhir saya,
2. bukan merupakan duplikasi karya tulis yang sudah dipublikasikan atau yang pernah dipakai untuk mendapatkan gelar sarjana di universitas lain, kecuali pada bagian-bagian sumber informasi dicantumkan dengan cara referensi yang semestinya, dan
3. bukan merupakan karya terjemahan dari kumpulan buku atau jurnal acuan yang tertera di dalam referensi pada karya tugas akhir saya.

Kalau terbukti saya tidak memenuhi apa yang telah dinyatakan di atas, maka karya tugas akhir ini batal.

Tangerang, 8 Januari 2018
Yang membuat pernyataan

Samuel Ivan Gunadi

UNIVERSITAS PELITA HARAPAN
FAKULTAS ILMU KOMPUTER

PERSETUJUAN DOSEN PEMBIMBING TUGAS AKHIR

REAL-TIME GPU-BASED SPH FLUID SIMULATION USING VULKAN AND OPENGL COMPUTE SHADERS

oleh

Nama	**: Samuel Ivan Gunadi**
NIM	**: 11220120018**
Program Studi	**: Teknik Informatika**
Konsentrasi	**: Rekayasa Piranti Lunak**

telah diperiksa dan disetujui untuk diajukan dan dipertahankan dalam sidang tugas akhir guna memperoleh gelar Sarjana Informatika Strata Satu pada program studi Teknik Informatika, Fakultas Ilmu Komputer di Universitas Pelita Harapan, Tangerang, Banten.

Tangerang, 8 Januari 2018
Menyetujui:

Pembimbing Utama	**Co-Pembimbing**
Dr. Pujianto Yugopuspito	Dr. David H. Hareva
Ketua Program Studi Teknik Informatika	**Pembantu Dekan Fakultas Ilmu Komputer**
Irene A. Lazarusli, S.Kom., M.T.	Hendra Tjahyadi, S.T., M.T., Ph.D.

UNIVERSITAS PELITA HARAPAN

FAKULTAS ILMU KOMPUTER

PERSETUJUAN TIM PENGUJI TUGAS AKHIR

Pada 30 Januari 2018 telah diselenggarakan sidang tugas akhir untuk memenuhi sebagian persyaratan akademik guna memperoleh gelar Sarjana Informatika Strata Satu pada program studi Teknik Informatika, Fakultas Ilmu Komputer, Universitas Pelita Harapan, atas nama

Nama	**: Samuel Ivan Gunadi**
NIM	**: 11220120018**
Program Studi	**: Teknik Informatika**
Fakultas	**: Ilmu Komputer**

termasuk ujian tugas akhir yang berjudul **REAL-TIME GPU-BASED SPH FLUID SIMULATION USING VULKAN AND OPENGL COMPUTE SHADERS** oleh tim penguji yang terdiri dari:

Nama Penguji	**Jabatan dalam Tim Penguji**	**Tanda Tangan**
1. Dr. Benny Hardjono	Ketua	________________
2. Dr. Pujianto Yugopuspito	Anggota	________________
3. Dr. David H. Hareva	Anggota	________________

ABSTRAK

Samuel Ivan Gunadi (11220120018)
REAL-TIME GPU-BASED SPH FLUID SIMULATION USING VULKAN AND OPENGL COMPUTE SHADERS
(xii + 41 halaman: 7 gambar; 2 tabel; 9 listing; 1 appendix)

Tidak seperti CPU yang bekerja di frekuensi yang sangat tinggi untuk mencapai kecepatan yang tinggi, GPU punya arsitektur parallel yang terdiri dari banyak streaming multiprocessors (SMs) yang bekerja di frekuensi yang lebih rendah, yang memperbolehkan efisiensi energi dan kecepatan eksekusi yang lebih tinggi jika algorithma yang dipakai dapat dibuat paralel. Karena metode smoothed particle hydrodynamics (SPH) punya skalabilitas yang bagus dan secara dasarnya dapat dibuat paralel, metode tersebut dapat memanfaatkan kekuatan pemrosesan paralel dari GPU dengan menggunakan Vulkan. Vulkan adalah application programming interface (API) grafis dan komputasi yang baru dari Khronos Group.

Tesis ini menginvestigasi potensi dari Vulkan untuk animasi fluida dengan metode SPH. Sebuah algoritma SPH paralel yang sederhana dibuat dan diimplementasikan dalam GLSL (OpenGL Shading Language) compute shader. Kode GLSL ini diubah menjadi format SPIR-V (Standard Portable Intermediate Representation V) lalu diimplementasikan dengan Vulkan dan OpenGL 4.6. Implementasi ini di verifikasi dengan dengan 2 tes kasus: menjatuhkan sebuah balok air ke dalam kotak dan bendungan pecah dalam saluran tertutup di temperatur ruang.

Bila jumlah partikelnya 30 000 atau lebih, implementasi Vulkan kami berjalan lebih cepat dibandingkan dengan implementasi OpenGL 4.6 kami ($n = 60000$, Vulkan = 28.4 fps, OpenGL 4.6 = 15.25 fps). Namun implementasi Vulkan kami berjalan lebih lambat dibandingkan dengan implementasi OpenGL bila jumlah partikelnya 20 000 atau kurang ($n = 20000$, Vulkan = 115.95 fps, OpenGL 4.6 = 120.8 fps; $n = 10000$, Vulkan = 271.8 fps, OpenGL 4.6 = 302.65 fps).

Referensi: 23 (1977-2017)

ABSTRACT

Samuel Ivan Gunadi (11220120018)

REAL-TIME GPU-BASED SPH FLUID SIMULATION USING VULKAN AND OPENGL COMPUTE SHADERS

(xii + 41 pages: 7 figures; 2 tables; 9 listings; 1 appendix)

Unlike CPU that works at very high frequency to achieve high speed, graphics processing units (GPUs) have a parallel architecture composed of many streaming multiprocessors (SMs) that work at a lower frequency, allowing for lower power consumption and faster execution time if the algorithm is able to be made parallel. Because the smoothed particle hydrodynamics (SPH) method has good scalability and is inherently parallelizable, it can take advantage of the parallel computing power of the GPUs by utilizing Vulkan. Vulkan is a new graphics and compute application programming interface (API) by Khronos Group.

This thesis investigates the potential of Vulkan for fluid animation using SPH method. A simple parallel SPH algorithm is devised and implemented in GLSL (OpenGL Shading Language) compute shader. This GLSL code is converted into SPIR-V (Standard Portable Intermediate Representation V) format and then implemented with Vulkan and OpenGL 4.6. These implementations are then verified using 2 test cases: dropping a cube of water into a box and a dam break in a closed channel at room temperature.

If the number of particles is 30 000 or greater, our Vulkan implementation is faster compared to our OpenGL implementation (n = 60000, Vulkan = 28.4 fps, OpenGL 4.6 = 15.25 fps). However, our Vulkan implementation is slower compared to our OpenGL 4.6 implementation if the number of particles is 20 000 or fewer (n = 20000, Vulkan = 115.95 fps, OpenGL 4.6 = 120.8 fps; n = 10000, Vulkan = 271.8 fps, OpenGL 4.6 = 302.65 fps).

References: 23 (1977-2017)

ACKNOWLEDGEMENTS

In no particular order, I would like to thank:

- Mr. Hendra Tjahyadi, S.T., M.T., Ph.D. as Associate Dean of Faculty of Computer Science.
- Ms. Irene A. Lazarusli, S.Kom., M.T. as Department Chair and Academic Advisor, for her constant help and lots of caring and patience in the past five years.
- Dr. Pujianto Yugopuspito as Thesis Advisor and Dr. David H. Hareva as Thesis Co-Advisor, for their patience, motivation, inspiration, enthusiasm, and invaluable guidance.
- my uncle, Mr. Tadius S. Gunadi, and my aunt, Dr. Lydia Pratanu, who have changed my life. Without them, this would not have been possible.
- my parents, for their unconditional love, encouragement, and support. Life has been hard on me, but they give me the strength to carry on.
- other family members for their support and confidence in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER I
INTRODUCTION

This chapter begins by providing the background knowledge. Next, the problem to be solved is defined. A brief review on the related fields is presented. The objectives and scope of this thesis are summarized. At the end of this chapter, the outline of this thesis is given.

1.1 Background

There are two common approaches to simulating fluid flows: Lagrangian approach (particle-based) and Eulerian approach (grid-based) [1, p. 134]. In Eulerian approach, the properties (such as pressure, density, and velocity) of fluid motion are represented by functions of space and time. The information about the flow is obtained in terms of what happens at fixed points in space as the fluid flows through those points.

The Lagrangian approach involves tracking individual fluid particles as they move about and determining how the fluid properties associated with these particles change as a function of time. That is, the fluid particles are labelled or identified, and their properties determined as they move. The smoothed particle hydrodynamics (SPH) method is based on Lagrangian approach.

1.1.1 Smoothed particle hydrodynamics method

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 [2] 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 [3, p. 116].

The *smoothness* in SPH is described using a function called kernel. When a particle position is given, this kernel function spreads out any values stored in the particle nearby. Starting from the center point of a particle, the function fades out to zero as the distance from the center reaches the kernel radius [3, p. 117].

1.1.2 GPU architecture

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) [4, p. 174].

1.1.3 OpenGL

OpenGL is an application programming interface (API), which is a software library for accessing features in graphics hardware. OpenGL is designed as a streamlined, hardware-independent interface that can be implemented on many different types of graphics hardware systems, or entirely in software—if no graphics hardware is present in the system—independent of a computer's operating or windowing system

[5].

OpenGL is implemented as a client-server system, with the application created by the developer being considered the client, and the OpenGL implementation provided by the manufacturer of your computer graphics hardware being the server [5].

1.1.4 Vulkan

Vulkan is a new graphics and compute application programming interface (API) by Khronos Group. The processor in a Vulkan device usually has many threads and so the computational model in Vulkan is heavily based on parallel computing. The Vulkan device also has access to device memory that may be shared with the main processor on which the application is running, and Vulkan also exposes this memory [6].

While OpenGL is based on a global state machine, Vulkan follows an object-oriented design without global state and objects are always manipulated directly with their handles. Compared to OpenGL, Vulkan is lower-level and more explicit and puts more responsibility on the application developer, and consequently the driver does less work. A driver is a program that takes the commands and data forming the API and translates them into something that the hardware can understand.

OpenGL driver does a lot in the background, from simple things such as state management, error checking, compiling high-level shaders, to avoiding deletion which operates asynchronously of resources that are still used by the GPU, or managing internal resource allocation and hardware cache flushing. Another example is the handling of out of memory situation, where the OpenGL driver implicitly splits up workloads or moves allocations between dedicated memory and system memory. While this is great for developers, it expends CPU time even when the application is running correctly. Vulkan addresses this by placing almost all state tracking, synchronization, and memory management into the hands of the application developer and by delegating correctness checks to *validation layers* [6]. These validation layers can be disabled for release builds and enabled for release builds.

1.1.5 Shader

A shader is a computer program that can be executed by the GPU. While OpenGL includes all the compiler tools internally to take the source code of the shader, Vulkan API requires the Standard Portable Intermediate Representation V (SPIR-V) format—a low-level binary intermediate representation (IR)—for all shaders. With the `ARB_gl_spirv` extension, or in OpenGL 4.6 core specification, OpenGL also accepts shaders in SPIR-V form much like it accepts shaders in GLSL form. Typically, for SPIR-V format, an offline tool chain will generate SPIR-V from a high-level shading language such as GLSL.

1.2 Problem definition

Simulating fluids like water, smoke, and fire using physics-based simulation is a very popular topic in computer graphics. One of the popular methods for doing fluid simulation on a computer is SPH, which is parallelizable and has good scalability. One way to exploit the computational power of massively parallel processors available in current GPUs is with Vulkan, the new compute and graphics API. And by the time of writing, SPH application in Vulkan is yet to be found.

1.3 Literature review

Fluid animations have been implemented with various physical models and hardware platforms. Jos Stam [7] developed a mesh-based solution for fluid animation. Nick

Foster and Ron Fedkiw [8] derived full 3D solution for Navier–Stokes equations that produced realistic animation results. In addition to the basic method, the Lagrangian equations of motion are used to place buoyant dynamic objects into a scene, and track the position of spray and foam during the animation process. Jos Stam [9] later extended the basic Eulerian approach by approximating the flow equations in order to achieve near real-time performance. He also demonstrated various special effects of fluid with simple C code at typical frame rate of 4 to 7 minutes per frame.

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.

Reeves [10] introduced the particle system which is then widely used to model the deformable bodies, clothes, and water. His paper demonstrated animation of fire and multicolored fireworks. Particle-based animations are created with two approaches, one with motion defined by certain physical model and other by simple use of Newton's basic laws. Gingold and Monaghan proposed *Smoothed Particle Hydrodynamics*, a particle based model to simulate astrophysical phenomena [2] and later extended to simulate free surface incompressible fluid flows [11]. This model was created for scientific analysis of fluid flow, carried out with few particles. Müller et al. extended the basic SPH method for fluid simulation for interactive application [12] and designed a new SPH kernel.

The first implementation of the SPH method totally on GPU was realized by Harada et al. [13] 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.

1.4 Objectives and scope

The main objectives of this thesis are:

1. Implement a real-time (greater than 60 frames per second) SPH simulation in Vulkan and OpenGL 4.6 using compute shaders. The targeted platforms are standard desktop and laptop computers.
2. Implement and develop rendering of the simulation.
3. Implement 2 test cases to verify that simulation behaves correctly, and then measure its performance. The first case is dropping a cube of water into a box. The second case is a dam break in a closed channel.

1.5 Thesis outline

The rest of this thesis is structured as follows. Chapter II discusses the theory of fluid dynamics and SPH method, and the algorithm for the simulation loop. Chapter III covers the challenges, strategies, and implementation for the GPU. Chapter IV shows the experimental results of the implementation using 2 test cases and the performance test results. The final chapter V summarizes the work presented in this thesis.

CHAPTER II
FLUID SIMULATION

The study of fluid simulation is known as Computational Fluid Dynamics, which encompasses mathematical models for fluid behavior and numerical methods for implementing these models. 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.

2.1 Navier–Stokes Equation

The Navier–Stokes equations for incompressible, isothermal Newtonian fluids with constant viscosity [14], [15] in vector notation are the momentum equation

$$\rho(\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v}) = -\nabla p + \rho \vec{g} + \mu \nabla^2 \vec{v} \tag{2.1}$$

or

$$\rho \frac{D\vec{v}}{Dt} = -\nabla p + \rho \vec{g} + \mu \nabla^2 \vec{v} \tag{2.2}$$

and the continuity equation

$$\nabla \cdot \vec{v} = 0. \tag{2.3}$$

Newtonian fluids are defined as fluids for which the shear stress is linearly proportional to the shear strain rate [1, p. 465]. Equation 2.1 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.

2.2 Smoothed particle hydrodynamics

The key idea behind smoothed particle hydrodynamics is the integral approximation of a field function [16]. The integral representation of a field function A is

$$A(r) = \int_{\omega} A(r')W(r - r', h)dr' \tag{2.4}$$

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

$$A(\vec{r}) = \sum_j A_j V_j W(\vec{r} - \vec{r}_j, h), \tag{2.5}$$

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

$$V = \frac{m}{\rho}. \tag{2.6}$$

Thus the particle approximation of a field function in SPH is

$$A(\vec{r}) = \sum_j A_j \frac{m_j}{\rho_j} W(\vec{r} - \vec{r}_j, h), \tag{2.7}$$

2.3 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 2.7

$$A(\vec{r}) = \sum_j \rho_j \frac{m_j}{\rho_j} W(\vec{r} - \vec{r}_j, h), \tag{2.8}$$

which is then simplified to

$$\rho_i = \sum_j m_j W(\vec{r}_i - \vec{r}_j, h). \tag{2.9}$$

2.4 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

$$P = R\rho T, \tag{2.10}$$

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

$$P = k\rho, \tag{2.11}$$

where k is a gas stiffness constant that depends on the temperature. As suggested by Desbrun [17], the equation is then modified to

$$P_i = k(\rho_i - \rho_0), \tag{2.12}$$

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

2.5 Computing acceleration

The acceleration of a fluid particle can be expressed as

$$\vec{a} = \frac{D\vec{v}}{Dt} = \frac{\partial\vec{v}}{\partial t} + \vec{v}\cdot\nabla\vec{v} = \frac{\partial\vec{v}}{\partial t} + u\frac{\partial\vec{v}}{\partial x} + v\frac{\partial\vec{v}}{\partial y} + w\frac{\partial\vec{v}}{\partial z}, \tag{2.13}$$

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

$$\vec{a}_i = \frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{\rho_i}, \tag{2.14}$$

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.

2.6 Computing forces

The momentum equation (Equation 2.1) can be separated into three components on the right hand side

$$\frac{d\vec{v}_i}{dt} = \vec{F}_i^{\text{pressure}} + \vec{F}_i^{\text{viscosity}} + \vec{F}_i^{\text{external}}. \tag{2.15}$$

The pressure force F_i^{pressure} can be computed by applying Equation 2.7 to the pressure term $-\nabla p$

$$\vec{F}_i^{\text{pressure}} = -\nabla p(\vec{r}_i) = -\sum_j m_j \frac{P_j}{\rho_j} \nabla W(\vec{r}_i - \vec{r}_j, h). \tag{2.16}$$

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]

$$\vec{F}_i^{\text{pressure}} = -\sum_j m_j \frac{P_i + P_j}{2\rho_j} \nabla W(\vec{r}_i - \vec{r}_j, h). \tag{2.17}$$

The viscosity force $F_i^{\text{viscosity}}$ can be computed by applying Equation 2.7 to the viscosity term $\mu \nabla^2 v$

$$\vec{F}_i^{\text{viscosity}} = \mu \nabla^2 \vec{v}(\vec{r}_i) = \mu \sum_j m_j \frac{\vec{v}_j}{\rho_j} \nabla^2 W(\vec{r}_i - \vec{r}_j, h). \tag{2.18}$$

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

$$\vec{F}_i^{\text{viscosity}} = \mu \sum_j m_j \frac{\vec{v}_j - \vec{v}_i}{\rho_j} \nabla^2 W(\vec{r}_i - \vec{r}_j, h). \tag{2.19}$$

2.7 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

$$\vec{v}_i{}' = \vec{v}_i + dt\frac{\vec{F}}{\rho_i} \tag{2.20}$$

$$\vec{r}_i{}' = \vec{r}_i + dt\ \vec{v}_i. \tag{2.21}$$

2.8 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.

2.8.1 Poly6 kernel

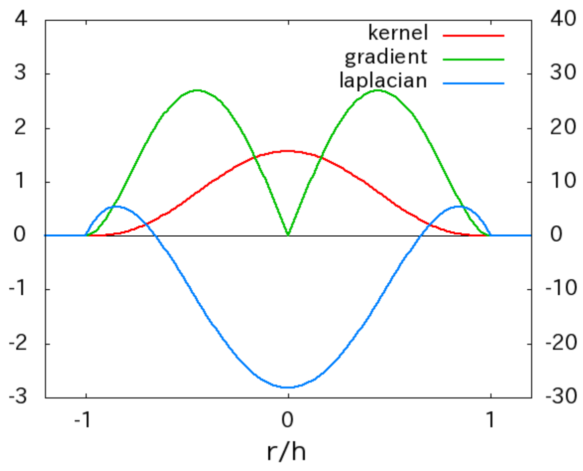


Figure 2.1: Plot of W_{poly6} kernel, retrieved from [18]

The poly6 kernel is also known as the sixth-degree polynomial kernel. The poly6 kernel is used for density computation (Equation 2.9).

$$W_{\text{poly6}}(\vec{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - \|\vec{r}\|^2)^3, & 0 \le \|\vec{r}\|^2 \le h \\ 0, & \text{otherwise} \end{cases} \tag{2.22}$$

The gradient of this kernel is used for surface normal.

$$\nabla W_{\text{poly6}}(\vec{r}, h) = -\frac{945}{32\pi h^9} \begin{cases} \vec{r}(h^2 - \|\vec{r}\|^2)^2, & 0 \le \|\vec{r}\|^2 \le h \\ 0, & \text{otherwise} \end{cases} \tag{2.23}$$

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

$$\nabla^2 W_{\text{poly6}}(\vec{r}, h) = -\frac{945}{32\pi h^9} \begin{cases} (h^2 - \|\vec{r}\|^2)(3h^2 - 7\|\vec{r}\|^2), & 0 \le \|\vec{r}\|^2 \le h \\ 0, & \text{otherwise} \end{cases} \tag{2.24}$$

2.8.2 Spiky kernel

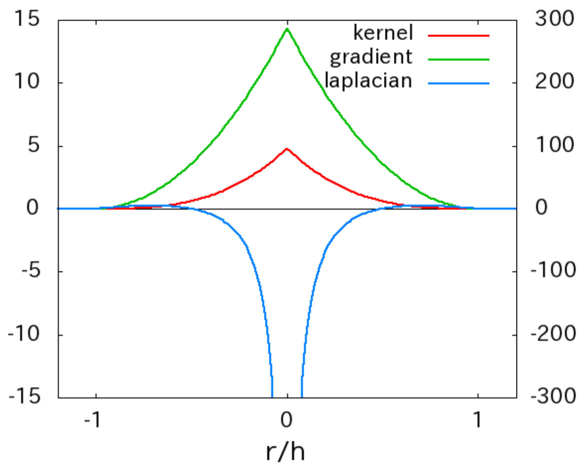


Figure 2.2: Plot of W_{spiky} kernel, retrieved from [18]

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 [17] is used

$$W_{\text{spiky}}(\vec{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - \|\vec{r}\|)^3, & 0 \leq \|\vec{r}\| \leq h \\ 0, & \text{otherwise} \end{cases} \tag{2.25}$$

The gradient of spiky kernel is used to compute the pressure force (Equation 2.17).

$$\nabla W_{\text{spiky}}(\vec{r}, h) = -\frac{45}{\pi h^6} \begin{cases} (h - \|\vec{r}\|)^2 \hat{r}, & 0 \leq \|\vec{r}\| \leq h \\ 0, & \text{otherwise} \end{cases} \tag{2.26}$$

2.8.3 Viscosity kernel

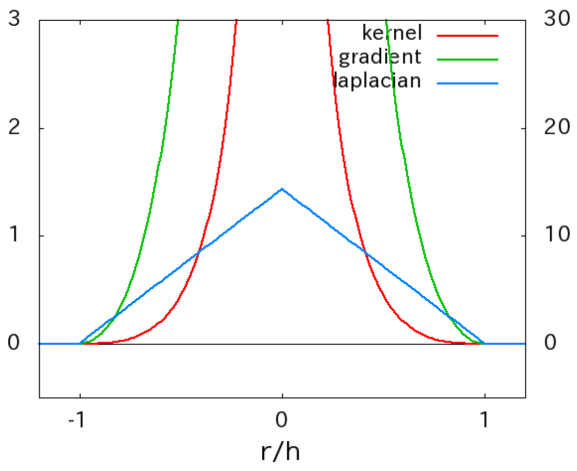


Figure 2.3: Plot of $W_{\text{viscosity}}$ kernel, retrieved from [18]

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.

$$W_{\text{viscosity}}(\vec{r}, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{\|\vec{r}\|^3}{2h^3} + \frac{\|\vec{r}\|^2}{h^2} + \frac{h}{2\|\vec{r}\|} - 1, & 0 \leq \|\vec{r}\| \leq h) \\ 0, & otherwise \end{cases} \tag{2.27}$$

The Laplacian of this kernel is used for computing viscosity force (Equation 2.19).

$$\nabla^2 W_{\text{viscosity}}(\vec{r}, h) = \frac{45}{\pi h^6}(h - \|\vec{r}\|) \tag{2.28}$$

2.9 Neighbor search

In SPH, it is important to efficiently and quickly find the neighboring particles in a given radius. The naïve 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.

2.10 Simulation loop

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

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
```

CHAPTER III
IMPLEMENTATION DETAILS

This chapter covers the details of implementing the SPH method in both Vulkan and OpenGL 4.6. The full source code is available online [19]–[21].

3.1 Data structures

The particle data structure contains all the attributes necessary for a particle: position, velocity, force, density, and pressure. In our implementation, the particle mass, radius, and resting density are the same for all particles so they are defined as constants.

We use a flat data structure (Listing 3.2). A data structure is flat if its 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 [22, p. 145]. We initially used `struct` (Listing 3.1), 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.

Listing 3.1: Old particle data structure

```
struct particle
{
    vec2 position;
    vec2 velocity;
    vec2 force;
    float density;
    float pressure;
};

layout(std430, binding = 0) buffer particles_block
{
    particle particles[];
};
```

Listing 3.2: New particle data structure

```
layout(std430, binding = 0) buffer position_block
{
    vec2 position[];
};

layout(std430, binding = 1) buffer velocity_block
{
    vec2 velocity[];
};

layout(std430, binding = 2) buffer force_block
{
    vec2 force[];
};

layout(std430, binding = 3) buffer density_block
{
    float density[];
};

layout(std430, binding = 4) buffer pressure_block
{
    float pressure[];
};
```

3.2 Compute shader

A compute shader is one of the shader stages, the other stages being the vertex shader, fragment shader, geometry shader, tessellation control shader, and tessellation evaluation shader. A compute shader's purpose is to carry out computations on data, which may be used further in the pipeline for rendering or for any other heavy calculations. 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.

3.3 SPIR-V format

All GLSL source files are compiled into SPIR-V first using Khronos Group's glslang. Note that both Vulkan and OpenGL implementations use the same GLSL code.

3.4 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. Based on our empirical data (Table 3.1), 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).

3.5 Buffers

Buffers are the primary structure to store data on graphics hardware. They represent the GPU's internal memory associated with everything from geometry, textures, and

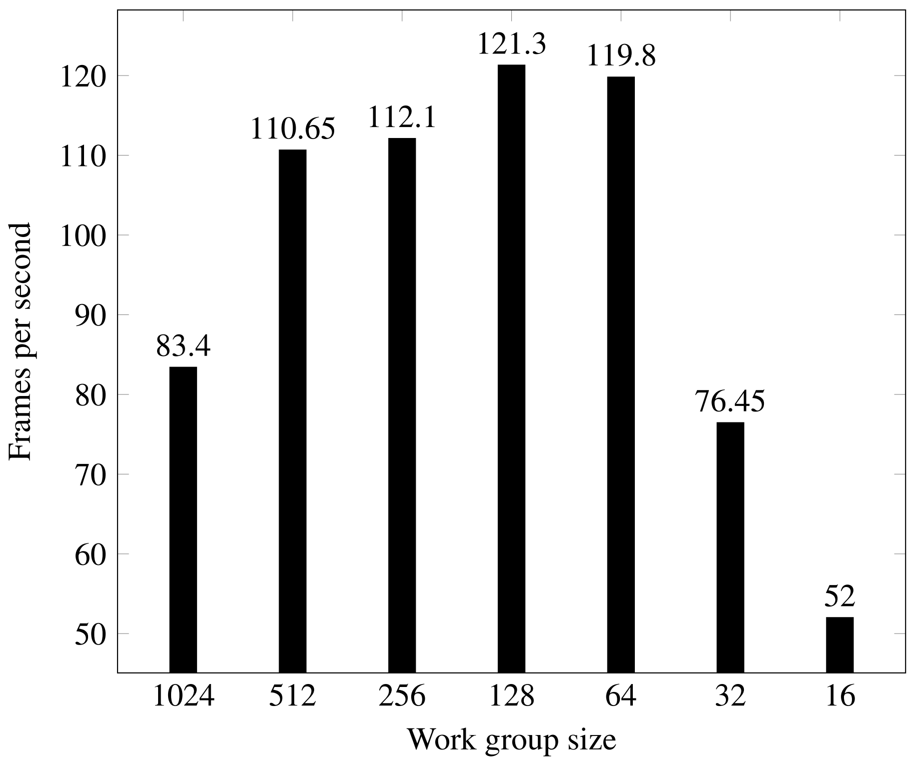


Figure 3.1: Frame rates of our 20 000-particle OpenGL implementation between different numbers of work group size

image plane data. From a programming standpoint, an application must initialize the buffers on the GPU that are needed for the application, which is a host-to-device copy operation. Device-to-host copies can be performed as well to pull data from the GPU to the CPU memory [23, p. 442]. OpenGL and Vulkan also allow device memory to be mapped into host memory so that an application program can write directly to the buffers on the graphics card. In our implementations, we use a single buffer which stores all the particles state which is used by both the compute pipelines and then reused by the graphics pipeline for rendering.

3.6 Shader storage buffer object

A shader storage buffer object (SSBO) is a buffer object that is used to store and retrieve data from within the GLSL. Five SSBOs are used in our implementations, which are bound to a single buffer with different offsets (Listing 3.3 and Listing 3.4).

The built-in inputs of compute shaders provide a useful way of accessing elements of a SSBO. For a program that runs through a given number of particles, using `gl_GlobalInvocation.x` in the shader will give the unique index of a particle.

Listing 3.3: Binding a single buffer to multiple indexed buffer binding points in Vulkan

```
    // ssbo sizes
    const uint64_t position_ssbo_size = sizeof(glm::vec2) * SPH_NUM_PARTICLES;
    const uint64_t velocity_ssbo_size = sizeof(glm::vec2) * SPH_NUM_PARTICLES;
    const uint64_t force_ssbo_size = sizeof(glm::vec2) * SPH_NUM_PARTICLES;
    const uint64_t density_ssbo_size = sizeof(float) * SPH_NUM_PARTICLES;
    const uint64_t pressure_ssbo_size = sizeof(float) * SPH_NUM_PARTICLES;

    const uint64_t packed_buffer_size = position_ssbo_size + velocity_ssbo_size + force_ssbo_size + density_ssbo_size + pressure_ssbo_size;
    // ssbo offsets
    const uint64_t position_ssbo_offset = 0;
    const uint64_t velocity_ssbo_offset = position_ssbo_size;
    const uint64_t force_ssbo_offset = velocity_ssbo_offset + velocity_ssbo_size;
    const uint64_t density_ssbo_offset = force_ssbo_offset + force_ssbo_size;
    const uint64_t pressure_ssbo_offset = density_ssbo_offset + density_ssbo_size;
```

```
    VkDescriptorBufferInfo descriptor_buffer_infos[]
    {
        {
            packed_particles_buffer_handle,
            position_ssbo_offset,
            position_ssbo_size
        },
        {
            packed_particles_buffer_handle,
            velocity_ssbo_offset,
            velocity_ssbo_size
        },
        {
            packed_particles_buffer_handle,
            force_ssbo_offset,
            force_ssbo_size
        },
        {
            packed_particles_buffer_handle,
            density_ssbo_offset,
            density_ssbo_size
        },
        {
            packed_particles_buffer_handle,
            pressure_ssbo_offset,
            pressure_ssbo_size
        }
    };
```

Listing 3.4: Binding a single buffer to multiple indexed buffer binding points in OpenGL

```
    // ssbo sizes
    constexpr ptrdiff_t position_ssbo_size = sizeof(glm::vec2) * SPH_NUM_PARTICLES;
    constexpr ptrdiff_t velocity_ssbo_size = sizeof(glm::vec2) * SPH_NUM_PARTICLES;
    constexpr ptrdiff_t force_ssbo_size = sizeof(glm::vec2) * SPH_NUM_PARTICLES;
    constexpr ptrdiff_t density_ssbo_size = sizeof(float) * SPH_NUM_PARTICLES;
    constexpr ptrdiff_t pressure_ssbo_size = sizeof(float) * SPH_NUM_PARTICLES;

    constexpr ptrdiff_t packed_buffer_size = position_ssbo_size + velocity_ssbo_size + force_ssbo_size + density_ssbo_size + pressure_ssbo_size;
    // ssbo offsets
    constexpr ptrdiff_t position_ssbo_offset = 0;
    constexpr ptrdiff_t velocity_ssbo_offset = position_ssbo_size;
    constexpr ptrdiff_t force_ssbo_offset = velocity_ssbo_offset + velocity_ssbo_size;
    constexpr ptrdiff_t density_ssbo_offset = force_ssbo_offset + force_ssbo_size;
    constexpr ptrdiff_t pressure_ssbo_offset = density_ssbo_offset + density_ssbo_size;
```

```
    // bindings
    glBindBufferRange(GL_SHADER_STORAGE_BUFFER, 0, packed_particles_buffer_handle, position_ssbo_offset, position_ssbo_size);
    glBindBufferRange(GL_SHADER_STORAGE_BUFFER, 1, packed_particles_buffer_handle, velocity_ssbo_offset, velocity_ssbo_size);
    glBindBufferRange(GL_SHADER_STORAGE_BUFFER, 2, packed_particles_buffer_handle, force_ssbo_offset, force_ssbo_size);
    glBindBufferRange(GL_SHADER_STORAGE_BUFFER, 3, packed_particles_buffer_handle, density_ssbo_offset, density_ssbo_size);
    glBindBufferRange(GL_SHADER_STORAGE_BUFFER, 4, packed_particles_buffer_handle, pressure_ssbo_offset, pressure_ssbo_size);
```

3.7 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 (Listing 3.5 and Listing 3.6). This ensures that previous shader has finished writing new data before reading it (read after write [RAW] dependency).

Listing 3.5: Dispatching 3 compute shaders in OpenGL with memory barriers for synchronization

```
    glUseProgram(compute_program_handle[0]);
    glDispatchCompute(SPH_NUM_WORK_GROUPS, 1, 1);
    glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
    glUseProgram(compute_program_handle[1]);
    glDispatchCompute(SPH_NUM_WORK_GROUPS, 1, 1);
    glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
    glUseProgram(compute_program_handle[2]);
    glDispatchCompute(SPH_NUM_WORK_GROUPS, 1, 1);
    glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Listing 3.6: Dispatching 3 compute shaders in Vulkan with pipeline barriers for synchronization

```
    if (vkBeginCommandBuffer(compute_command_buffer_handle, &command_buffer_begin_info) != VK_SUCCESS)
    {
        throw std::runtime_error("command buffer begin failed");
    }

    vkCmdBindDescriptorSets(compute_command_buffer_handle, VK_PIPELINE_BIND_POINT_COMPUTE, compute_pipeline_layout_handle, 0, 1, &compute_descriptor_set_handle, 0, nullptr);

    // First dispatch
    vkCmdBindPipeline(compute_command_buffer_handle, VK_PIPELINE_BIND_POINT_COMPUTE, compute_pipeline_handles[0]);
    vkCmdDispatch(compute_command_buffer_handle, SPH_NUM_WORK_GROUPS, 1, 1);

    // Barrier: compute to compute dependencies
    // First dispatch writes to a storage buffer, second dispatch reads from that storage buffer
    vkCmdPipelineBarrier(compute_command_buffer_handle, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, 0, 0, nullptr, 0, nullptr, 0, nullptr);

    // Second dispatch
    vkCmdBindPipeline(compute_command_buffer_handle, VK_PIPELINE_BIND_POINT_COMPUTE, compute_pipeline_handles[1]);
    vkCmdDispatch(compute_command_buffer_handle, SPH_NUM_WORK_GROUPS, 1, 1);

    // Barrier: compute to compute dependencies
    // Second dispatch writes to a storage buffer, third dispatch reads from that storage buffer
    vkCmdPipelineBarrier(compute_command_buffer_handle, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, 0, 0, nullptr, 0, nullptr, 0, nullptr);

    // Third dispatch
    // Third dispatch writes to the storage buffer. Later, vkCmdDraw reads that buffer as a vertex buffer with vkCmdBindVertexBuffers.
    vkCmdBindPipeline(compute_command_buffer_handle, VK_PIPELINE_BIND_POINT_COMPUTE, compute_pipeline_handles[2]);
    vkCmdDispatch(compute_command_buffer_handle, SPH_NUM_WORK_GROUPS, 1, 1);

    vkCmdPipelineBarrier(compute_command_buffer_handle, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, 0, 0, nullptr, 0, nullptr, 0, nullptr);

    vkEndCommandBuffer(compute_command_buffer_handle);
```

3.8 Simulation parameters

The kernel functions are originally designed for 3D, but our implementations are 2D. So the simulation parameters are adjusted to give better results.

- particle radius $r = 0.005$,

- initial inter-particle distance $dx = 2r$,
- particle mass $m = 0.02$,
- resting density $\rho_0 = 1000$,
- smoothing length $h = 4r$,
- time step $dt = 0.001$,
- stiffness $k = 2000$, and
- viscosity $\mu = 3000$.

3.9 Initializing particles

All particles attributes are initialized with 0, and then their locations are set depending on which test case. OpenGL initialize the buffer implicitly by calling `glBufferStorage`. In Vulkan, it is done explicitly using a staging buffer.

3.10 Visualization

The fluid is visualized as a point cloud. In OpenGL, this is done by simply passing `GL_POINTS` to the draw call, while in Vulkan it is done by specifying `VK_PRIMITIVE_TOPOLOGY_POINT_LIST` during the graphics pipeline creation.

CHAPTER IV
RESULTS

4.1 Test cases

Two test cases are used to verify that the implementations are correct by examining the visual output. The frame rates of our Vulkan and OpenGL implementations are shown in Figure 4.1 and Table 4.1. The first test case is dropping a cube of water inside a box, and the second is a dam break in a closed channel. The water is at rest initially (all velocities zero). The rendering of the first test case and the second test case is shown in Figure 4.2 and Figure 4.3 respectively.

4.2 Test environment

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.

4.3 Performance analysis

Figure 4.1 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 [23, p. 438]. 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.

The OpenGL API statistics from 20-second runtime (00:01 to 00:21) collected using NVIDIA Nsight are shown in Table 4.2. The total OpenGL API time is 98.4 %. Approximately 97.76 % of time is spent on computation and approximately 0.64 % is spent on rendering. At the time of writing, NVIDIA Nsight does not support Vulkan profiling, so the statistics for Vulkan are unavailable.

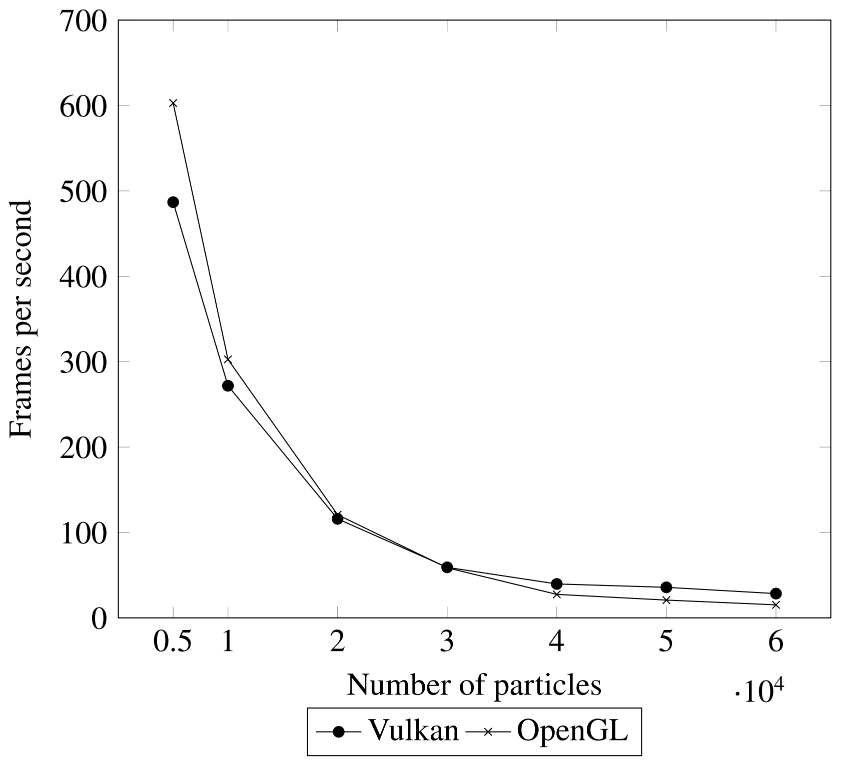


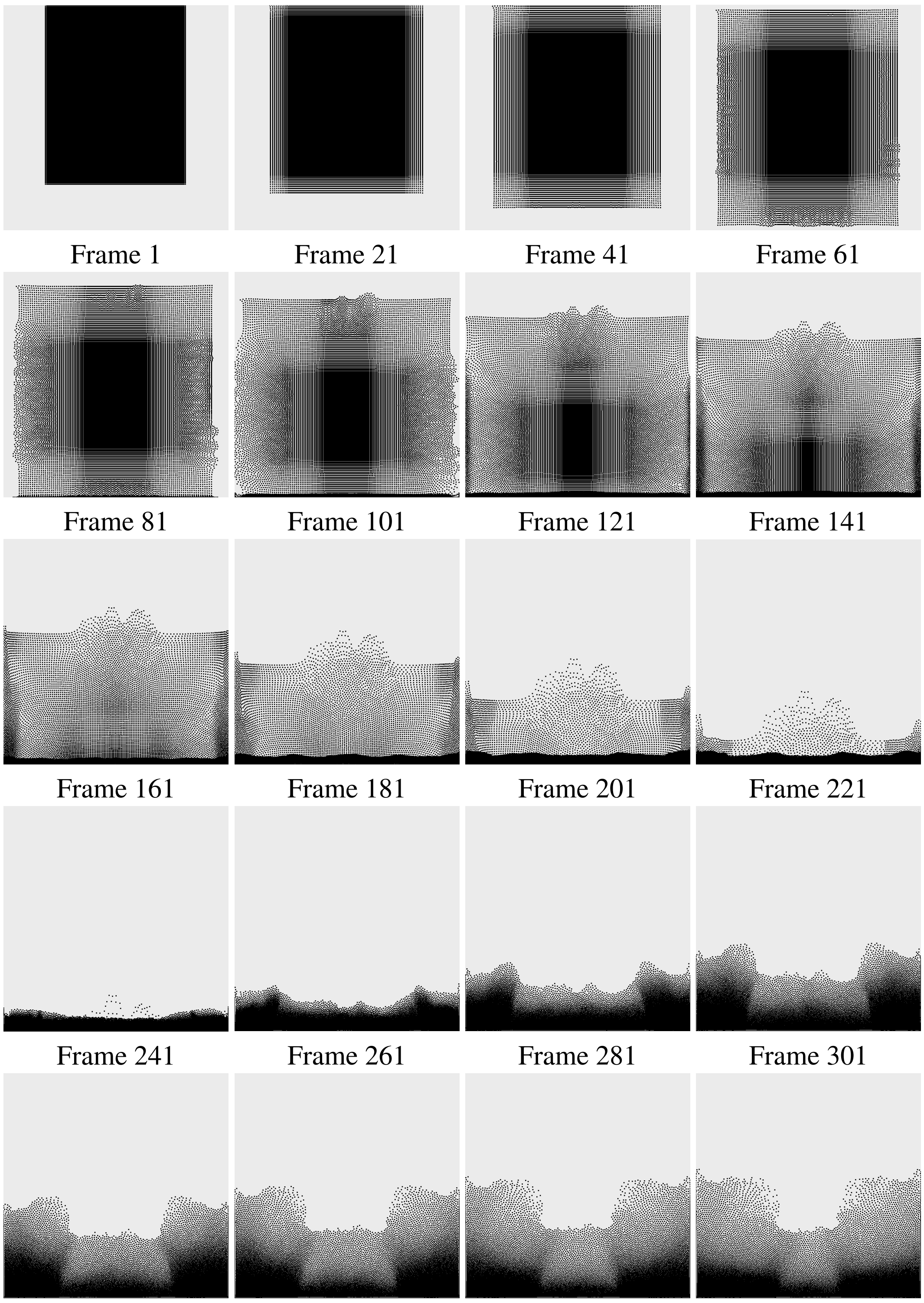
Figure 4.1: Frame rates comparison

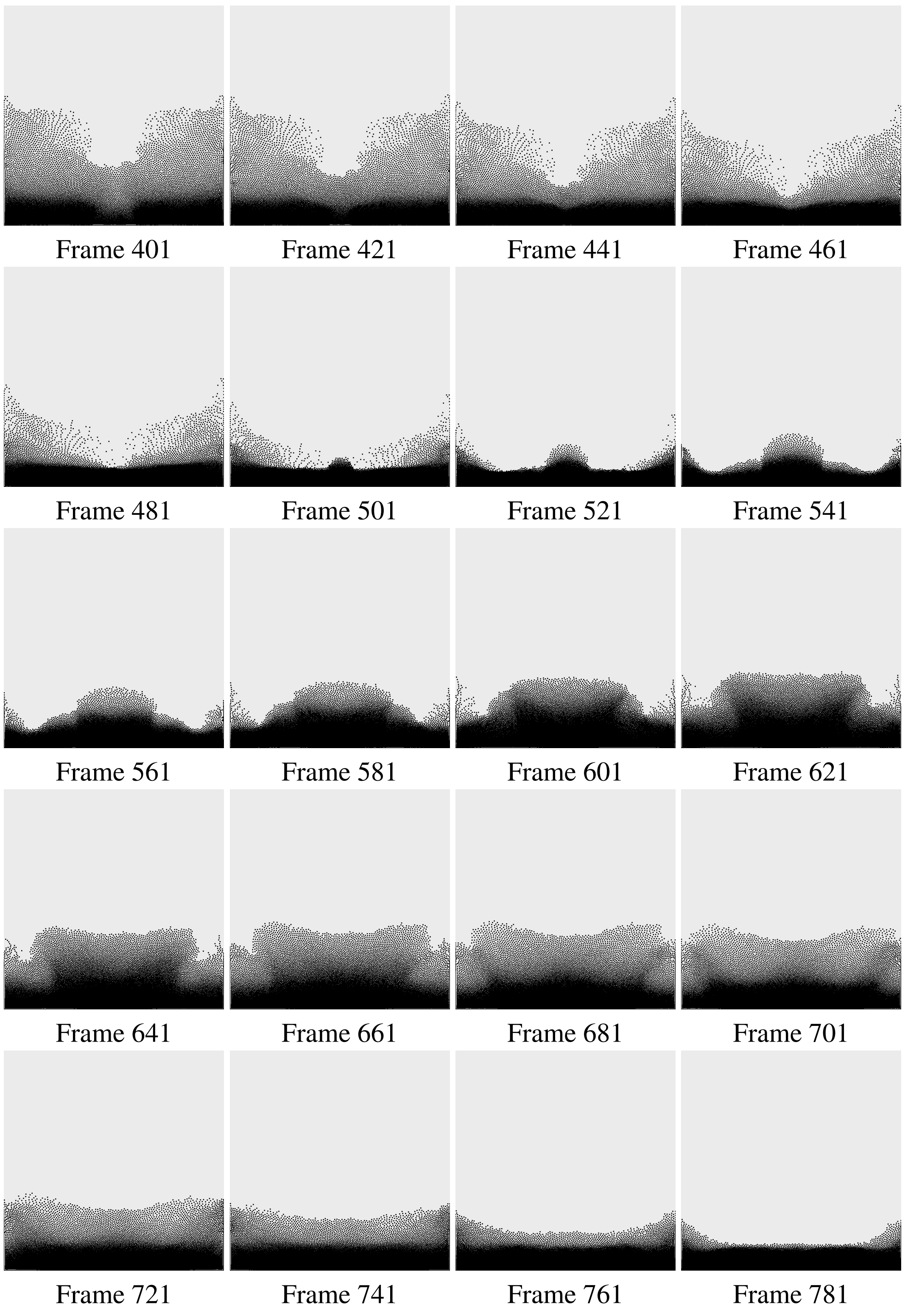
Table 4.1: Frame rates table

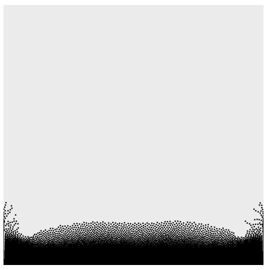
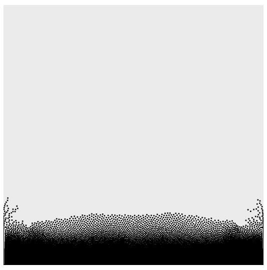
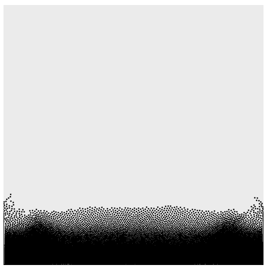
Number of particles	Vulkan (fps)	OpenGL (fps)
5000	486.80	602.90
10000	271.80	302.65
20000	115.95	120.80
30000	59.15	58.65
40000	39.70	27.50
50000	35.75	20.80
60000	28.40	15.25

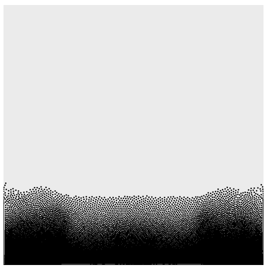
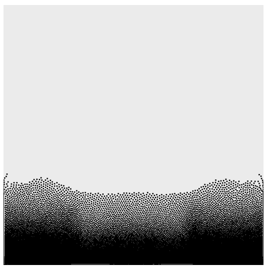
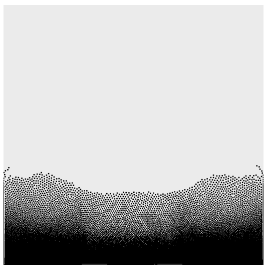
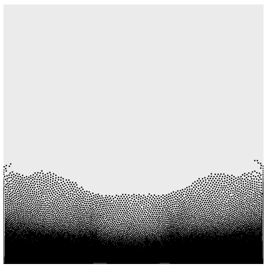
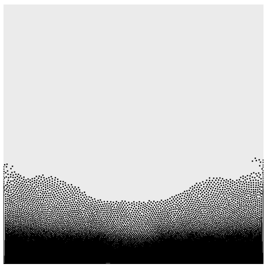
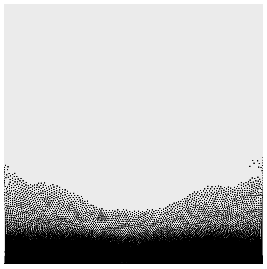
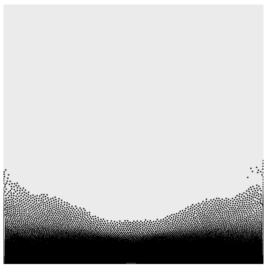
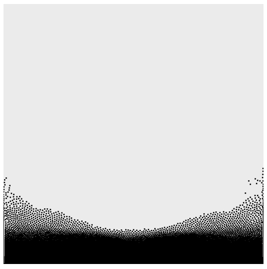
Table 4.2: OpenGL API calls statistics collected from 20 seconds of runtime

Name	Count	Time (µs))	Time (%)	Min (µs)	Avg (µs)	Max (µs)
`glDispatchCompute`	7539	19965136.886	95.87	2.640	2648.247	24859.006
`glMemoryBarrier`	7539	374533.469	1.80	0.138	49.679	20221.785
`glDrawArrays`	2514	67960.646	0.33	14.054	27.032	106.307
`SwapBuffers`	2514	47220.282	0.23	8.994	18.782	273.259
`glUseProgram`	10054	31403.814	0.15	0.127	3.123	24811.032
`GetPixelFormat`	2514	7734.347	0.04	1.680	3.076	90.057
`glClear`	2514	1783.426	0.01	0.269	0.709	25.131


Frame 1
Frame 21
Frame 41
Frame 61
Frame 81
Frame 101
Frame 121
Frame 141
Frame 161
Frame 181
Frame 201
Frame 221
Frame 241
Frame 261
Frame 281
Frame 301
Frame 321
Frame 341
Frame 361
Frame 381

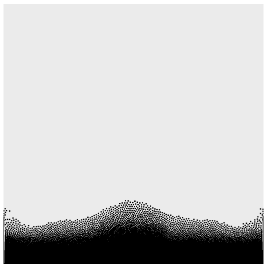
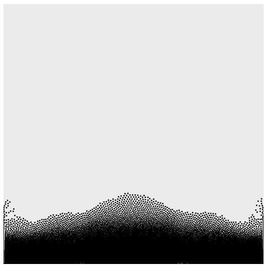

Frame 401
Frame 421
Frame 441
Frame 461
Frame 481
Frame 501
Frame 521
Frame 541
Frame 561
Frame 581
Frame 601
Frame 621
Frame 641
Frame 661
Frame 681
Frame 701
Frame 721
Frame 741
Frame 761
Frame 781

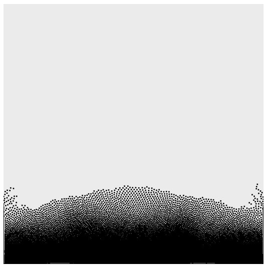
Frame 801

Frame 821

Frame 841

Frame 861

Frame 881

Frame 901

Frame 921

Frame 941

Frame 961

Frame 981

Frame 1001

Frame 1021

Frame 1041

Frame 1061

Frame 1081

Frame 1101

Frame 1121

Frame 1141

Frame 1161

Frame 1181

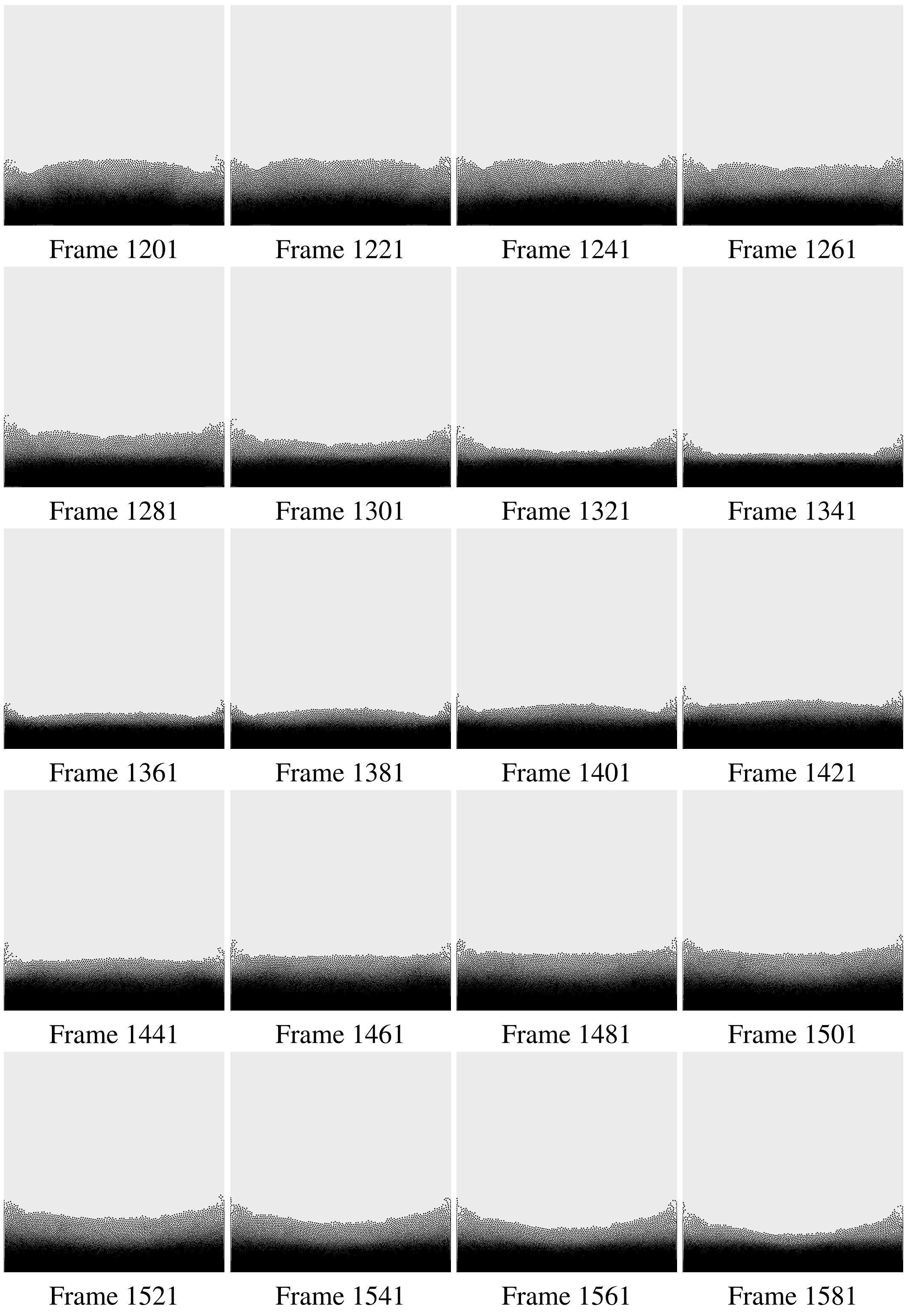
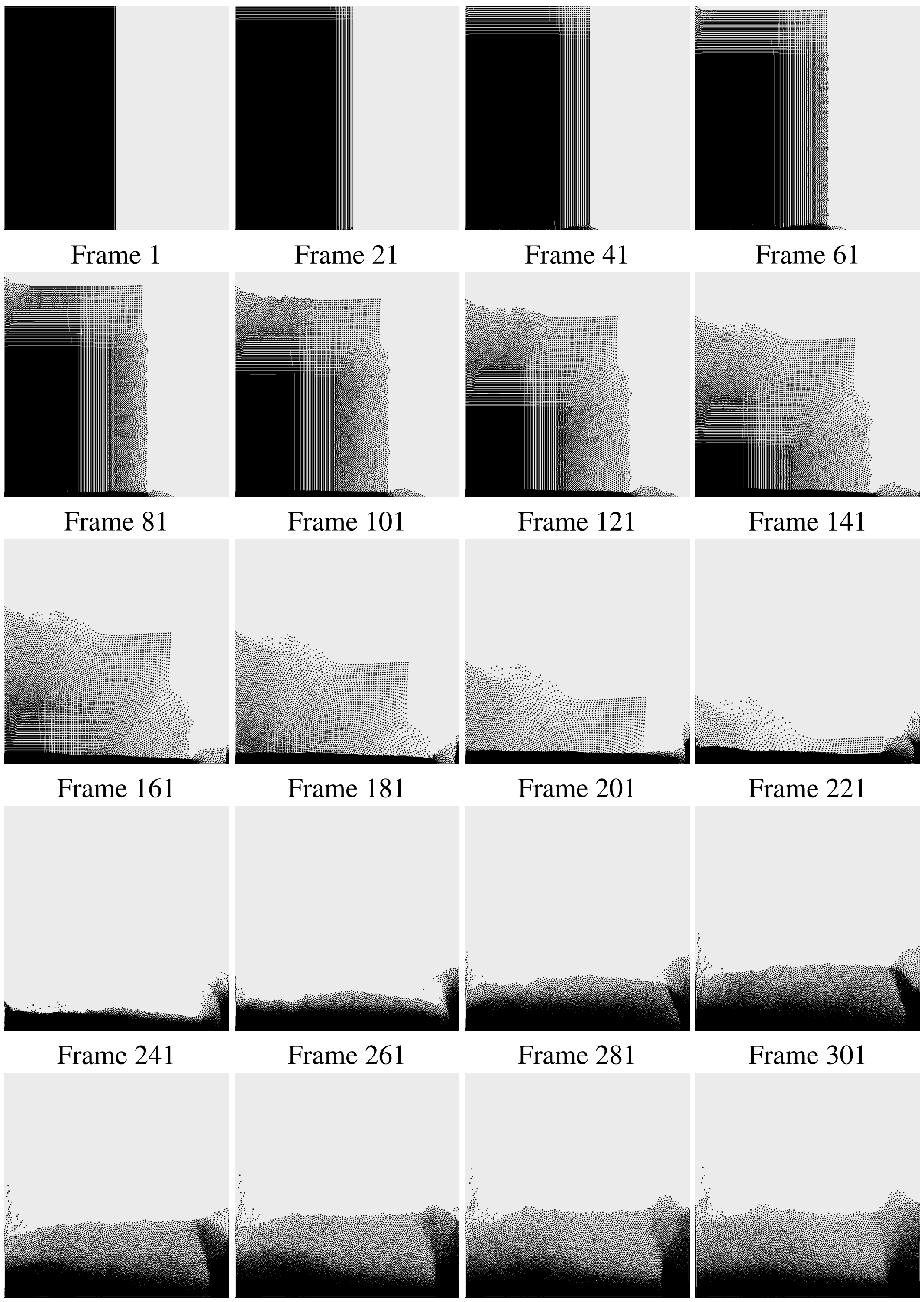
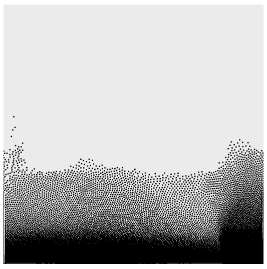
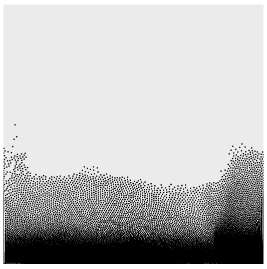
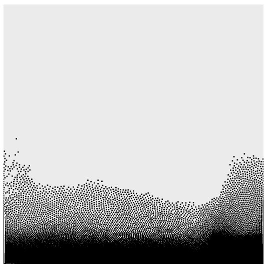
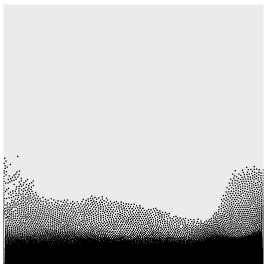
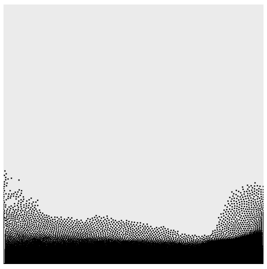
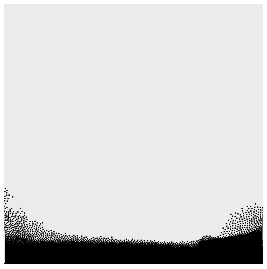
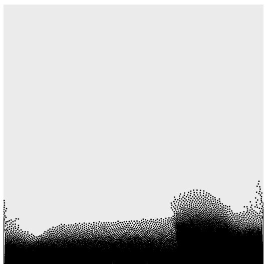
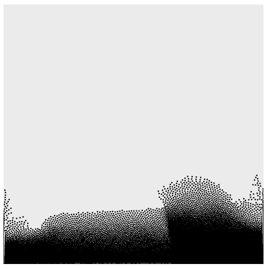


Figure 4.2: Rendering of the first test case


Frame 1
Frame 21
Frame 41
Frame 61
Frame 81
Frame 101
Frame 121
Frame 141
Frame 161
Frame 181
Frame 201
Frame 221
Frame 241
Frame 261
Frame 281
Frame 301
Frame 321
Frame 341
Frame 361
Frame 381

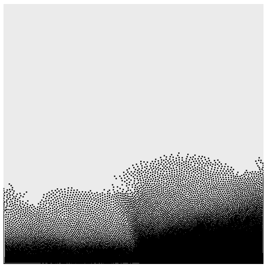
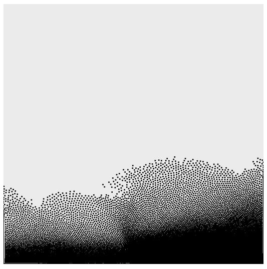

Frame 401

Frame 421

Frame 441

Frame 461

Frame 481

Frame 501

Frame 521

Frame 541

Frame 561

Frame 581

Frame 601

Frame 621

Frame 641

Frame 661

Frame 681

Frame 701

Frame 721

Frame 741

Frame 761

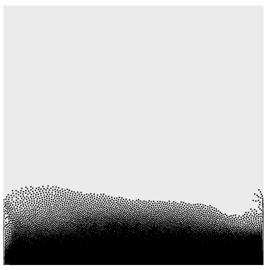
Frame 781

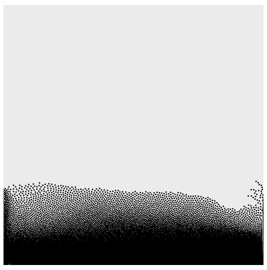
Frame 801

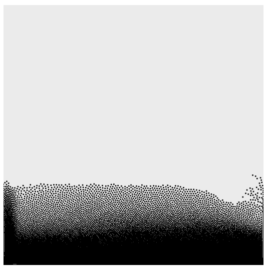
Frame 821

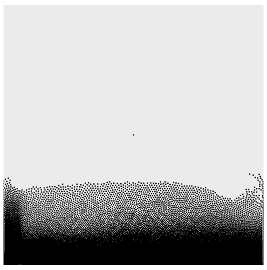
Frame 841

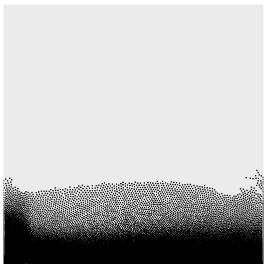
Frame 861


Frame 881

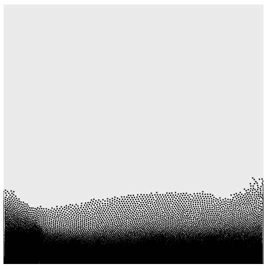

Frame 901

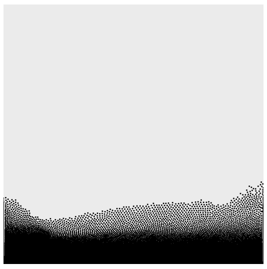

Frame 921


Frame 941


Frame 961

Frame 981

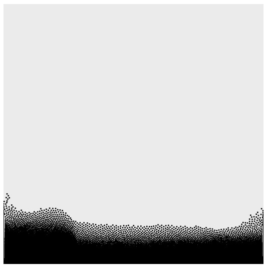

Frame 1001


Frame 1021

Frame 1041

Frame 1061

Frame 1081


Frame 1101

Frame 1121

Frame 1141

Frame 1161

Frame 1181

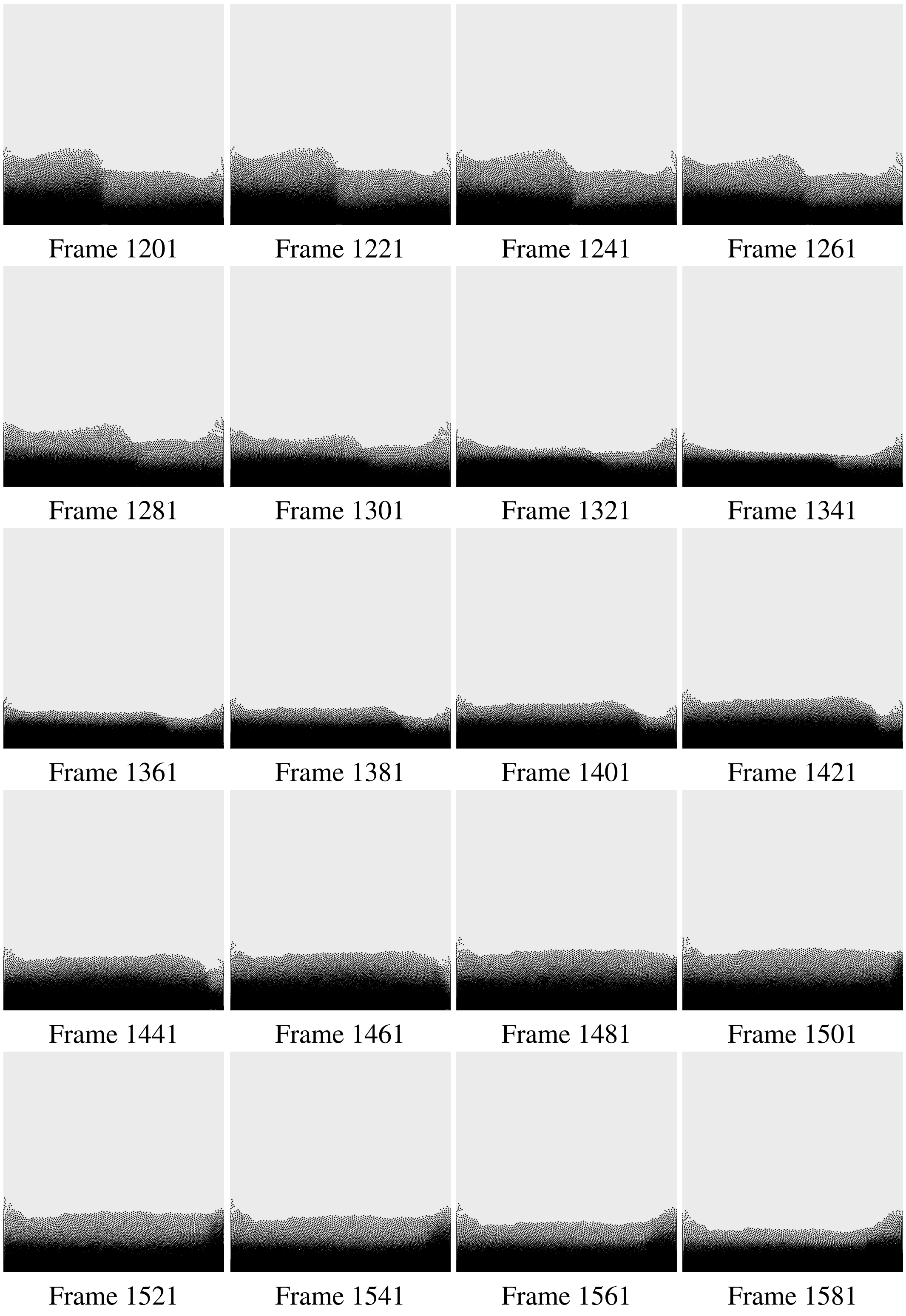


Figure 4.3: Rendering of the second test case

CHAPTER V
CONCLUSIONS

In this thesis, 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 which use the same GLSL code 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.

BIBLIOGRAPHY

[1] Y. A. Çengel and J. M. Cimbala, *Fluid mechanics, Fundamentals and applications*, 3rd ed. New York, NY, USA: McGraw-Hill Education, 2014, ISBN: 978-0-07-338032-2.

[2] 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. DOI: `10.1093/mnras/181.3.375`. eprint: `/oup/backfile/content_public/journal/mnras/181/3/10.1093/mnras/181.3.375/2/mnras181-0375.pdf`. [Online]. Available: `http://dx.doi.org/10.1093/mnras/181.3.375`.

[3] D. Kim, *Fluid engine development*. Boca Raton, FL, USA: CRC Press, 2017, ISBN: 978-1-4987-1992-6.

[4] Y. Cai and S. See, Eds., *Gpu computing and applications*. Singapore: Springer, 2015, ISBN: 978-981-287-133-6.

[5] J. Kessenich, G. Sellers, and D. Shreiner, *Opengl programming guide, The official guide to learning opengl, version 4.5 with spir-v*, 9th ed. Boston, MA, USA: Addison-Wesley Professional, 2016, ISBN: 978-0-13-449549-1.

[6] G. Sellers and J. Kessenich, *Vulkan programming guide: The official guide to learning vulkan*. Boston, MA, USA: Addison-Wesley Professional, 2016, ISBN: 978-0-13-446454-1.

[7] J. Stam, "Stable fluids," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '99, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128, ISBN: 0-201-48560-5. DOI: `10.1145/311535.311548`. [Online]. Available: `http://dx.doi.org/10.1145/311535.311548`.

[8] N. Foster and R. Fedkiw, "Practical animation of liquids," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01, New York, NY, USA: ACM, 2001, pp. 23–30, ISBN: 1-58113-374-X. DOI: `10.1145/383259.383261`. [Online]. Available: `http://doi.acm.org/10.1145/383259.383261`.

[9] J. Stam, "Real-time fluid dynamics for games," in *Proceedings of the game developer conference*, vol. 18, 2003, p. 25.

[10] W. T. Reeves, "Particle systems—a technique for modeling a class of fuzzy objects," *ACM Trans. Graph.*, vol. 2, no. 2, pp. 91–108, Apr. 1983, ISSN: 0730-0301. DOI: `10.1145/357318.357320`. [Online]. Available: `http://doi.acm.org/10.1145/357318.357320`.

[11] J. J. Monaghan, “Simulating free surface flows with sph,” *J. Comput. Phys.*, vol. 110, no. 2, pp. 399–406, Feb. 1994, ISSN: 0021-9991. DOI: `10.1006/jcph.1994.1034`. [Online]. Available: `http://dx.doi.org/10.1006/jcph.1994.1034`.

[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, San Diego, California: Eurographics Association, 2003, pp. 154–159, ISBN: 1-58113-659-5. [Online]. Available: `http://dl.acm.org/citation.cfm?id=846276.846298`.

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

[14] B. R. Munson, T. H. Okiishi, W. W. Huebsch, and A. P. Rothmayer, *Fundamentals of fluid mechanics*, 7th ed. Hoboken, NJ, USA: Wiley, 2013, ISBN: 978-1-118-11613-5.

[15] P. M. Gerhart, A. L. Gerhart, and J. I. Hochstein, *Munson, young and okiishi’s fundamentals of fluid mechanics*, 8th ed. Hoboken, NJ, USA: Wiley, 2016, ISBN: 978-1-119-08070-1.

[16] 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`.

[17] M. Desbrun and M.-P. Gascuel, “Smoothed particles: A new paradigm for animating highly deformable bodies,” in *Computer Animation and Simulation ’96: Proceedings of the Eurographics Workshop in Poitiers, France, August 31–September 1, 1996*, R. Boulic and G. Hégron, Eds. Vienna: Springer Vienna, 1996, pp. 61–76, ISBN: 978-3-7091-7486-9. DOI: `10.1007/978-3-7091-7486-9_5`. [Online]. Available: `https://doi.org/10.1007/978-3-7091-7486-9_5`.

[18] PukiWiki, *Sph - pukiwiki for pbcg lab*. [Online]. Available: `http://www.slis.tsukuba.ac.jp/~fujisawa.makoto.fu/cgi-bin/wiki/index.php?SPH%CB%A1%A4%CE%BD%C5%A4%DF%B4%D8%BF%F4` (visited on 2017).

[19] S. I. Gunadi, *Sph simulation in vulkan compute shader*. [Online]. Available: `https://github.com/multiprecision/sph_vulkan`.

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

[21] S. I. Gunadi, *Fast gpu-based sph fluid simulation using vulkan and opengl compute shaders*. [Online]. Available: `https://github.com/multiprecision/undergraduate_thesis`.

[22] K. Guntheroth, *Optimized c++, Proven techniques for heightened performance*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2016, ISBN: 978-1-4919-2206-4.

[23] S. Marschner and P. Shirley, *Fundamentals of computer graphics*, 4th ed. Natick, MA, USA: A. K. Peters, Ltd., 2016, ISBN: 978-1-4822-2939-4.

APPENDIX A
COMPLETE GLSL LISTINGS

Listing A.1: Compute shader 1—compute density and pressure

```glsl
#version 460

#define WORK_GROUP_SIZE 128

layout (local_size_x = WORK_GROUP_SIZE) in;

// constants
#define NUM_PARTICLES 20000

#define PI_FLOAT 3.1415927410125732421875f
#define PARTICLE_RADIUS 0.005f
#define PARTICLE_RESTING_DENSITY 1000
// Mass = Density * Volume
#define PARTICLE_MASS 0.02
#define SMOOTHING_LENGTH (4 * PARTICLE_RADIUS)

#define PARTICLE_STIFFNESS 2000

layout(std430, binding = 0) buffer position_block
{
    vec2 position[];
};

layout(std430, binding = 1) buffer velocity_block
{
    vec2 velocity[];
};

layout(std430, binding = 2) buffer force_block
{
    vec2 force[];
};

layout(std430, binding = 3) buffer density_block
{
    float density[];
};

layout(std430, binding = 4) buffer pressure_block
{
    float pressure[];
};

void main()
{
    uint i = gl_GlobalInvocationID.x;

    // compute density
    float density_sum = 0.f;
    for (int j = 0; j < NUM_PARTICLES; j++)
    {
        vec2 delta = position[i] - position[j];
        float r = length(delta);
        if (r < SMOOTHING_LENGTH)
        {
            density_sum += PARTICLE_MASS * /* poly6 kernel */ 315.f *       ↵
            pow(SMOOTHING_LENGTH * SMOOTHING_LENGTH - r * r, 3) / (64.f *   ↵
            PI_FLOAT * pow(SMOOTHING_LENGTH, 9));
        }
    }
    density[i] = density_sum;
    // compute pressure
    pressure[i] = max(PARTICLE_STIFFNESS * (density_sum -                  ↵
    PARTICLE_RESTING_DENSITY), 0.f);
}
```

Listing A.2: Compute shader 2—compute forces

```
#version 460

#define WORK_GROUP_SIZE 128

layout (local_size_x = WORK_GROUP_SIZE) in;

// constants
#define NUM_PARTICLES 20000

#define PI_FLOAT 3.1415927410125732421875f
#define PARTICLE_RADIUS 0.005f
#define PARTICLE_RESTING_DENSITY 1000
// Mass = Density * Volume
#define PARTICLE_MASS 0.02
#define SMOOTHING_LENGTH (4 * PARTICLE_RADIUS)

#define PARTICLE_VISCOSITY 3000.f

// OpenGL y-axis is pointing up, while Vulkan y-axis is pointing down.
// So in OpenGL this is negative, but in Vulkan this is positive.
#define GRAVITY_FORCE vec2(0, -9806.65)

layout(std430, binding = 0) buffer position_block
{
    vec2 position[];
};

layout(std430, binding = 1) buffer velocity_block
{
    vec2 velocity[];
};

layout(std430, binding = 2) buffer force_block
{
    vec2 force[];
};

layout(std430, binding = 3) buffer density_block
{
    float density[];
};

layout(std430, binding = 4) buffer pressure_block
{
    float pressure[];
};

void main()
{
    uint i = gl_GlobalInvocationID.x;
    // compute all forces
    vec2 pressure_force = vec2(0, 0);
    vec2 viscosity_force = vec2(0, 0);

    for (uint j = 0; j < NUM_PARTICLES; j++)
    {
        if (i == j)
        {
            continue;
        }
        vec2 delta = position[i] - position[j];
        float r = length(delta);
        if (r < SMOOTHING_LENGTH)
        {
            pressure_force -= PARTICLE_MASS * (pressure[i] + pressure[j]) / ↵
            (2.f * density[j]) *
            // gradient of spiky kernel
                -45.f / (PI_FLOAT * pow(SMOOTHING_LENGTH, 6)) *          ↵
                pow(SMOOTHING_LENGTH - r, 2) * normalize(delta);
            viscosity_force += PARTICLE_MASS * (velocity[j] - velocity[i]) /↵
            density[j] *
            // Laplacian of viscosity kernel
                45.f / (PI_FLOAT * pow(SMOOTHING_LENGTH, 6)) *           ↵
                (SMOOTHING_LENGTH - r);
        }
    }
    viscosity_force *= PARTICLE_VISCOSITY;
    vec2 external_force = density[i] * GRAVITY_FORCE;

    force[i] = pressure_force + viscosity_force + external_force;
}
```

Listing A.3: Compute shader 3—integrate and handle collision with the edges

```
#version 460

#define WORK_GROUP_SIZE 128

layout (local_size_x = WORK_GROUP_SIZE) in;

// constants
#define NUM_PARTICLES 20000

#define TIME_STEP 0.0001f
#define WALL_DAMPING 0.3f

layout(std430, binding = 0) buffer position_block
{
    vec2 position[];
};

layout(std430, binding = 1) buffer velocity_block
{
    vec2 velocity[];
};

layout(std430, binding = 2) buffer force_block
{
    vec2 force[];
};

layout(std430, binding = 3) buffer density_block
{
    float density[];
};

layout(std430, binding = 4) buffer pressure_block
{
    float pressure[];
};

void main()
{
    uint i = gl_GlobalInvocationID.x;

    // integrate
    vec2 acceleration = force[i] / density[i];
    vec2 new_velocity = velocity[i] + TIME_STEP * acceleration;
    vec2 new_position = position[i] + TIME_STEP * new_velocity;

    // boundary conditions
    if (new_position.x < -1)
    {
        new_position.x = -1;
        new_velocity.x *= -1 * WALL_DAMPING;
    }
    else if (new_position.x > 1)
    {
        new_position.x = 1;
        new_velocity.x *= -1 * WALL_DAMPING;
    }
    else if (new_position.y < -1)
    {
        new_position.y = -1;
        new_velocity.y *= -1 * WALL_DAMPING;
    }
    else if (new_position.y > 1)
    {
        new_position.y = 1;
        new_velocity.y *= -1 * WALL_DAMPING;
    }

    velocity[i] = new_velocity;
    position[i] = new_position;
}
```

Listing A.4: Vertex shader

```
#version 460

layout (location = 0) in vec2 position;

out gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
};

void main ()
{
    gl_Position = vec4(position.x, position.y, 0, 1);
    gl_PointSize = 5;
}
```

Listing A.5: Fragment shader

```
#version 460

layout(location = 0) out vec4 color;

void main ()
{
    color = vec4(0, 0, 0, 1);
}
```