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, syn-

chronization, 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 implementa-

tions 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.