Dumoulin Glenn

Go with the Flow: Making a deterministic fluid simulation in GPU-based environments

Supervisor: Vanden Abeele Alex

Coach: Geeroms Kasper

Graduation Work 2024-2025

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT & KEY WORDS

Fluid simulations are a common technique to visualize realistic fluid behaviour in virtual environments, and these simulations often benefit heavily from GPU acceleration. However, when these GPU-based simulations require deterministic results, things start to become a lot more difficult. This paper delves into this particular problem by attempting to employ a fixed-point arithmetic solution to address the non-deterministic nature of floating-point arithmetic on the GPU. The focus of this paper is to analyse the impact the fixed-point arithmetic implementation has on the performance and accuracy of the fluid simulation. This paper presents a partial solution for deterministic fluid simulations in a GPU-based environment by using fixed-point rounding.

Vloeistofsimulaties zijn een veelgebruikte techniek om realistisch vloeistofgedrag in virtuele omgevingen te visualiseren, en deze simulaties profiteren vaak sterk van GPU-versnelling. Wanneer deze GPU-gebaseerde simulaties echter deterministische resultaten vereisen, wordt het een stuk complexer. Dit artikel gaat dieper in op dit specifieke probleem door te proberen een oplossing te bieden met fixed-point-rekenkunde om de niet-deterministische aard van floating-point-rekenkunde op de GPU aan te pakken. De focus van dit artikel ligt op het analyseren van de impact van de implementatie van fixed-point-rekenkunde op de prestaties en nauwkeurigheid van de vloeistofsimulatie. Dit artikel presenteert een gedeeltelijke oplossing voor deterministische vloeistofsimulaties in een GPU-gebaseerde omgeving door middel van fixed-point-afronding.

Key words: Fluid simulation, GPU-based environment, determinism, fixed-point arithmetic

PREFACE

During my studies of game development, there has always been a topic that interested me but never seemed to get covered during classes, that topic being networking. Networking is a very active topic in the game industry today, and when this bachelor thesis presented itself on my to-do list the decision to create a research project around this topic by far the easiest decision of the project as a whole.

A while ago, I had the opportunity to go to Los Angeles, organized by our school, with a group of fellow students to visit several companies. It was during this trip that I had a conversation with a gameplay developer about possible ideas for a research topic around networking, and he suggested to do something with networked physics.

At this point, I got assigned a supervisor and coach to assist me during the process, and together with my supervisor the original topic for this research project was defined, I was going to synchronize a GPU-based fluid simulation over a network. This topic covered some networking experience for me and the fluid simulation was a nice way to include the networked physics in there as well.

However, I encountered many struggles during the early stages of the case study for this project and the final deadline kept on inching closer every day. Which eventually led to a much-needed discussion with my supervisor to rescope the project. This rescope sadly meant that the central reason behind this whole project, the networking, would have to be dropped from the project to allow for a successful completion of the project in the end.

I still want to gain some networking experience, but this research project will not be that opportunity anymore. Maybe I will revisit my original research topic when I have some spare time or maybe my upcoming internship will give me the opportunity to work in a networked environment. Only time will tell, and I am always looking forward to discovering what the future holds.

LIST OF FIGURES & TABLES

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Determinism in games is something that has been used for a while. Usually, this has been used to cover deterministic physics in networked environments to ensure all connected players always see and interact with the same physics simulation during a game session. However, research has been severely lacking when we require these deterministic results in a GPU-based environment.

The current approach to handle non-deterministic results, should floating-point arithmetic itself not provide good enough results based on the specific needs for your project, is to implement and use fixed-point arithmetic instead of floating-point arithmetic. Fixed-point values represent floating-point values as integers by scaling them by a certain factor depending on the required precision for the project.

This paper aims to implement this fixed-point arithmetic approach in a GPU-based environment. The environment that I will be using during this research project is a fluid simulation because of how heavily they benefit from GPU-acceleration, making them an ideal choice for this project. We will measure and discuss the impact fixed-point arithmetic has on the accuracy and performance of the simulation and compare it to the results using floating-point arithmetic.

RESEARCH QUESTION

The research question this paper tries to answer is the following:
“How does the implementation of fixed-point arithmetic impact the performance and accuracy of a fluid simulation in a GPU-accelerated environment?”

HYPOTHESES

The hypotheses that have been defined for this research project are as follows:

PERFORMANCE

The method for measuring performance will be explained in the Research chapter of this paper. The value used for defining “significant” is based on the performance results using floating-point arithmetic.

H0: Implementing fixed-point arithmetic has no significant impact on the performance (computation time) of the simulation. Where “no significant impact” will be defined as less than a 5% increase in computation time when compared to floating-point results.

H1: Implementing fixed-point arithmetic significantly impacts performance, reducing computational efficiency due to conversions between floating-point and fixed-point representations. Where “significantly impacts” will be defined as at least a 5% increase in computation time when compared to floating-point results.

ACCURACY

The method for measuring accuracy takes a sample from the simulation state and compares it against exact replicas from that simulation. This method will be explained in more detail in the Research chapter of this paper.

H0: Implementing fixed-point arithmetic has no significant effect on ensuring deterministic results for the GPU-accelerated fluid simulation, with results remaining nearly equal to those using floating-point arithmetic. Where “no significant effect” will be defined as less than a 50% decrease in differences between measurement iterations.

H1: Implementing fixed-point arithmetic guarantees deterministic results for the GPU-accelerated fluid simulation, resulting in exact matches between measurement iterations.

H2: Implementing fixed-point arithmetic reduces the accuracy of the simulation, with results being deterministic between measurement iterations but less precise in comparison to those using floating-point arithmetic.

LITERATURE STUDY / THEORETICAL FRAMEWORK

1. FLUID SIMULATIONS

Fluid simulation is a technique used to replicate and visualize the behaviour of fluids (liquids and gases) in a virtual environment. These simulations have a wide range of applications, from visual effects for movies and games to scientific research in fields such as aerodynamics and weather simulations. [1]

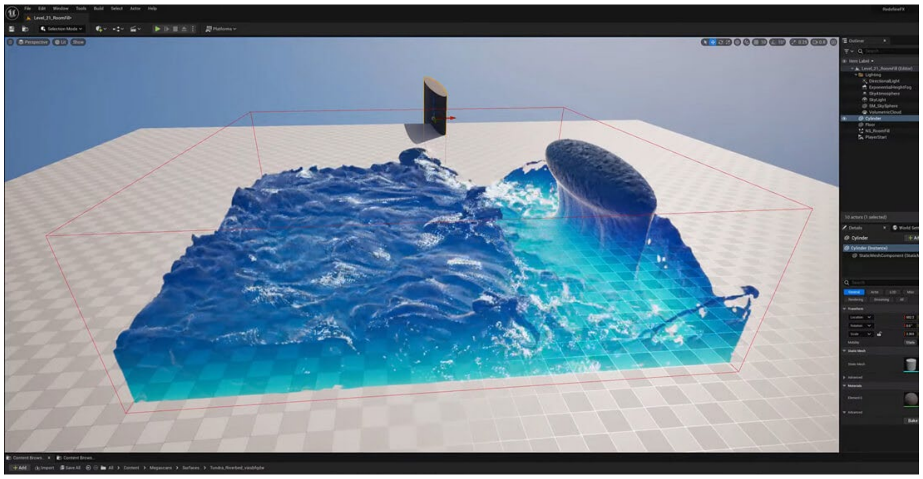


Figure 1: Example of 3D fluid simulation in Unreal Engine (Jesse Pitela)

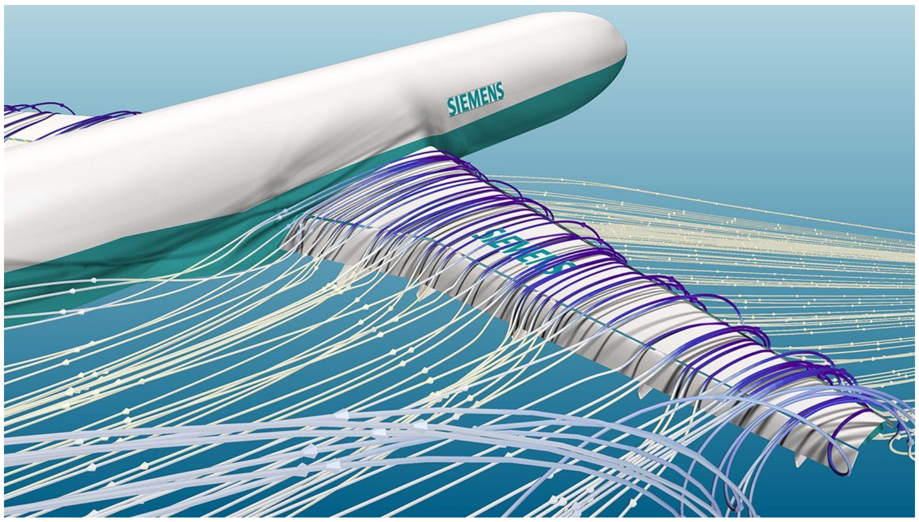


Figure 2: Example of Computational Fluid Dynamics for aerodynamics (Ascend Tech)

The primary goal of a fluid simulation is to represent the dynamics of fluid flow in a realistic manner. This is achieved by solving mathematical models that describe the movement of fluid particles and the forces acting upon them. Among these models, the Navier-Stokes equations play a pivotal role.

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Figure 3: The Navier-Stokes Equations for the velocity in a compact vector notation (top) and the equation for a density moving through the velocity field (bottom) (Jos Stam)

The Navier-Stokes equations are partial differential equations which describe the motion of viscous fluid substances. They lay the foundation to most fluid simulations and are derived from the principles of conservation of mass, momentum, and energy. [2] In summary this means that the equations ensure that mass is neither created nor destroyed within the fluid and account for forces like viscosity, pressure, and external forces (such as gravity).

Over the years, several methods have been developed to simulate fluid dynamics. We will take a look at two approaches that are actively being used, these approaches are particle-based and grid-based simulations.

1.1. PARTICLE-BASED SIMULATIONS

In particle-based simulations the fluid is represented by individual particles that interact with each other and their environment. Every particle is a small portion of the fluid and manages its own position and velocity as it moves through the fluid. A possible method used for particle-based simulations is the Smoothed Particle Hydrodynamics (SPH) method by R. A. Gingold and J. J. Monaghan. [3]

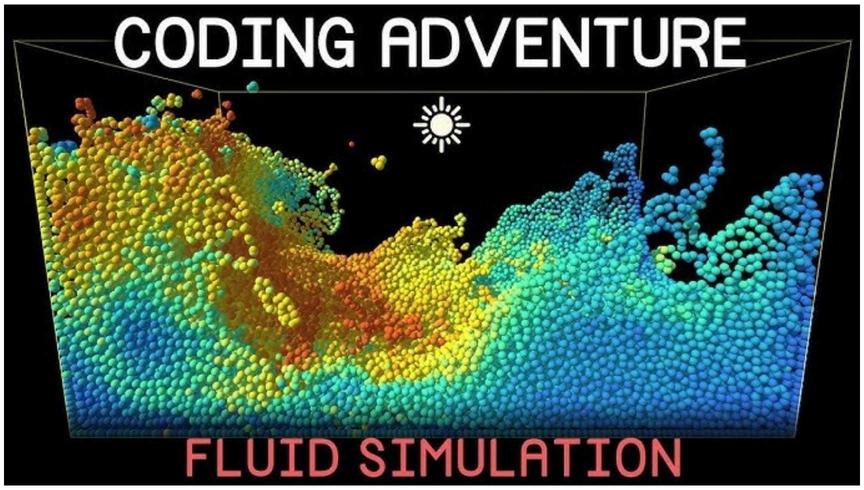


Figure 4: Example of particle-based fluid simulation using SPH (Sebastian Lague)

In SPH, fluid properties, such as density, pressure, and velocity, are calculated based on the surrounding particles, creating a physically accurate approximation of fluid flow. SPH also guarantees conservation of mass without extra computation since the particles themselves represent mass. Particle-based methods are especially effective for simulating free-surface flows, such as water in a container, or for applications requiring realistic visual effects, like splashes, waves, and interactions with solid objects. [4], [5]

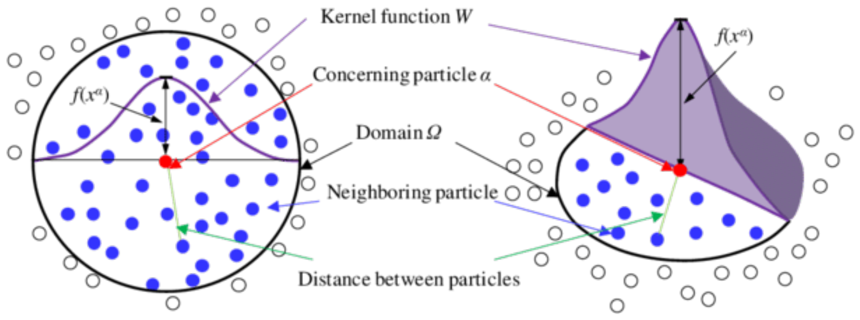


Figure 5: The concept of Smoothed Particle Hydrodynamics (Zili Dai, Fawu Wang, Yu Huang, Kun Song, Akinori Lio)

One drawback over grid-based techniques is the need for large numbers of particles to produce simulations of equivalent resolution. However, accuracy can be significantly higher with sophisticated grid-based techniques since it is easier to enforce the incompressibility condition in these systems. SPH for fluid simulation is being used increasingly in real-time animation and games where accuracy is not as critical as interactivity. [4]

1.2. GRID-BASED SIMULATIONS

In grid-based simulations the fluid is divided into a fixed grid of cells, and each grid cell contains the fluid properties (such as velocity, pressure, and density) at that point in the fluid simulation. The simulation is updated iteratively for each time step, updating the properties of the fluid for every grid cell.

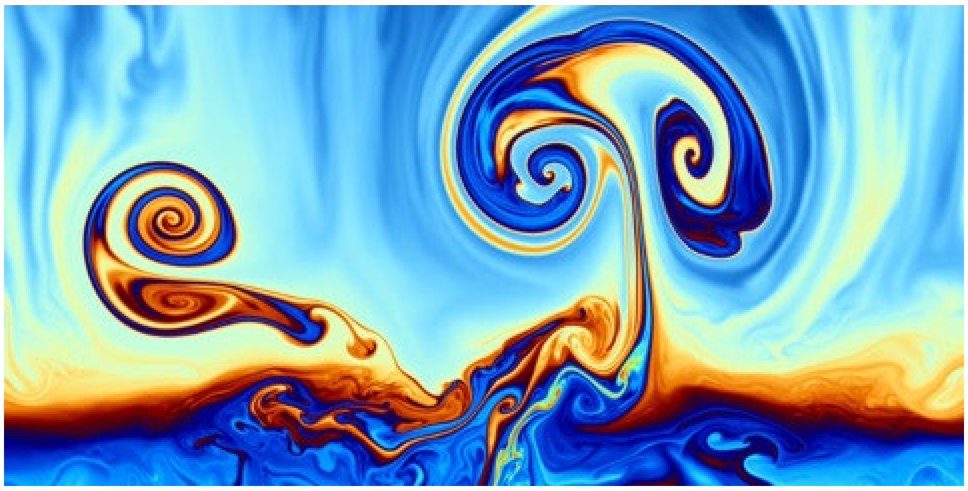


Figure 6: Example of grid-based fluid simulation using Stable Fluids (Karl Sims)

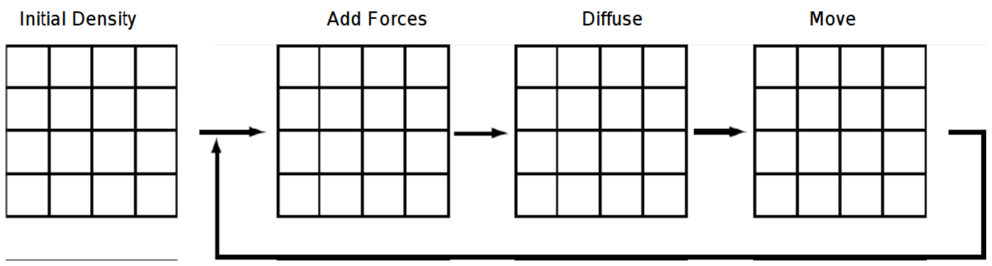


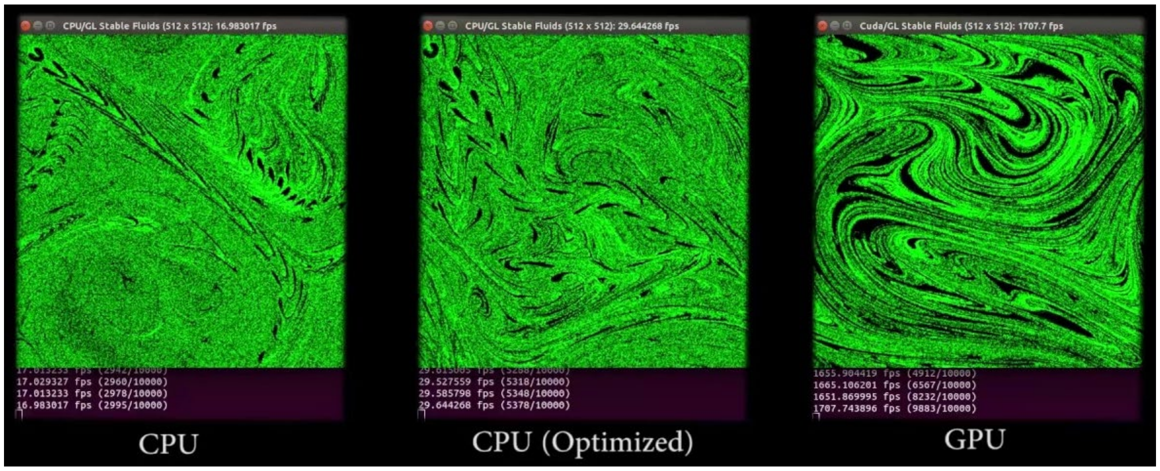
Figure 7: Basic structure of the density solver. At every time step we resolve the three terms appearing on the right hand side of the density equation (see bottom of Figure 3) (Jos Stam)

These methods are typically more stable and efficient than particle-based simulations, particularly for large-scale simulations. A possible technique for grid-based simulations is the "Stable Fluids" algorithm by J. Stam. [6], [7] This approach is used in both fluid dynamics research and computer graphics, especially for simulating fluids in constrained environments (such as rivers, oceans, or fluid flows within a confined space).

The “Stable Fluids” approach is based on the Navier-Stokes equations. These equations are notoriously hard to solve when strict physical accuracy is of prime importance. The “Stable Fluids” solvers on the other hand are geared towards visual quality. The emphasis is on stability and speed, which means that the simulations can be advanced with arbitrary time steps. [6], [7]

1.3. GPU ACCELERATION

Both particle-based and grid-based simulations can benefit significantly from GPU (Graphics Processing Unit) acceleration. GPUs are designed to handle large amounts of parallel computations, which makes them particularly well-suited for the computationally demanding tasks required by fluid simulations. [8] By offloading the bulk of the simulation calculations to the GPU, the simulation can run much faster, enabling real-time fluid dynamics in interactive applications such as video games, virtual environments, or simulations for scientific visualization.


CPU/GL Stable Fluids (512 x 512): 16.983017 fps
CPU/GL Stable Fluids (512 x 512): 29.644268 fps
Cuda/GL Stable Fluids (512 x 512): 1707.7 fps
CPU
CPU (Optimized)
GPU

```
17.029327 fps (2960/10000)
17.013233 fps (2978/10000)
16.983017 fps (2995/10000)
```

```
29.527559 fps (5318/10000)
29.585798 fps (5348/10000)
29.644268 fps (5378/10000)
```

```
1655.904419 fps (4912/10000)
1665.106201 fps (6567/10000)
1651.869995 fps (8232/10000)
1707.743896 fps (9883/10000)
```

Figure 8: Benchmarks for a fluid simulation on CPU (~17fps) vs optimized CPU (~30fps) vs GPU (~1650fps) (Jesús Martín Berlanga)

In practice, grid-based fluid simulations are more commonly accelerated on the GPU because the regular structure of the grid lends itself well to parallelization. For instance, calculating the velocity and pressure fields for each grid point can be done in parallel, dramatically speeding up the simulation process. Particle-based methods can also be accelerated on the GPU, but the irregular structure of particles can make parallelization more challenging, requiring more complex algorithms for efficient execution. [9]

As fluid simulation techniques continue to evolve, the need for real-time, high-quality simulations in interactive environments will only increase, making GPU acceleration a critical aspect of fluid dynamics research and applications.

2. DETERMINISM IN GPU-BASED ENVIRONMENTS

This chapter will shortly explain determinism, apply it to floating-point arithmetic, and introduce a solution to guarantee deterministic calculations in a GPU-based environment.

2.1. DETERMINISM

Determinism refers to the property a system in which no randomness is involved in the development of future states of the system. A deterministic model will thus always produce the same output from a given starting condition or initial state. [10]

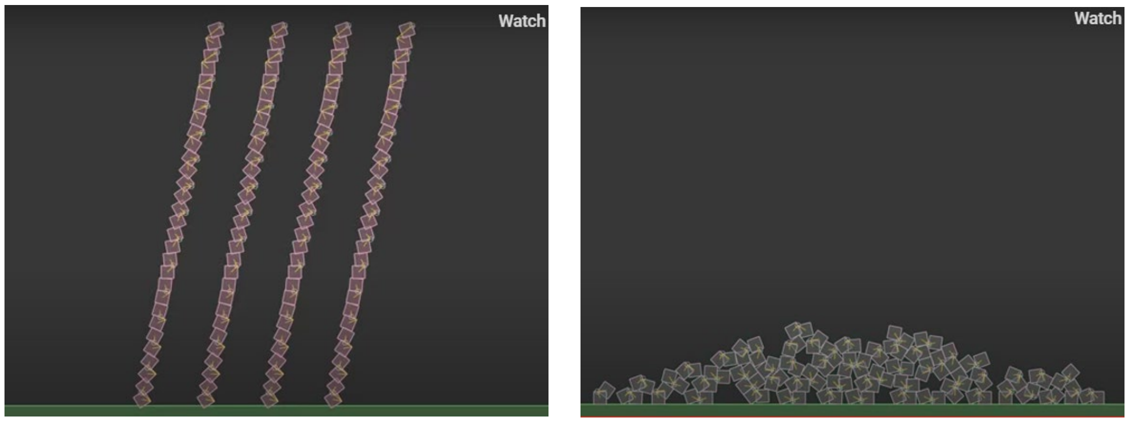


Figure 9: Example of the initial state (left) and final state (right) of a deterministic model in Box2D (Erin Catto) [11]

GPU-based environments, while highly performant, pose unique challenges to achieving determinism due to the non-deterministic nature of certain operations and the architecture of GPUs. [12]

2.2. NON-DETERMINISTIC FLOATING-POINT ARITHMETIC

Floating-point arithmetic, the default representation for real numbers, is inherently non-deterministic due to several factors. The most well-known is the presence of rounding errors, introducing minute discrepancies that accumulate over time. This factor is something that we can work around. However, there are other factors that are a lot more difficult to resolve. [13]

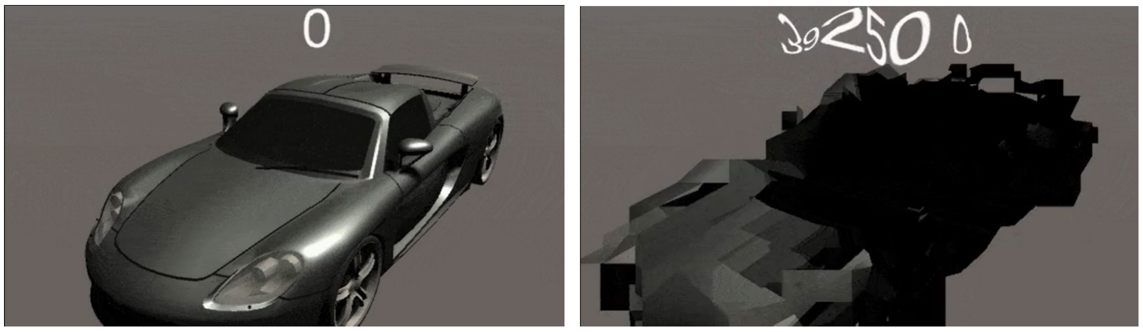


Figure 10: Example of floating-point precision errors for a fast moving object on the world origin (left) vs far from the world origin (right) (Alan Zucconi) [14]

One such factor is that there is no standardized implementation for floating-point arithmetic on the GPU. This means that different GPU manufacturers (such as NVIDIA and AMD) may implement floating-point standards differently. Even within the same manufacturer differences between GPU models can affect floating-point precision and execution order. [13]

2.3. FIXED-POINT ARITHMETIC

To address the non-deterministic nature of floating-point computations, we can employ fixed-point arithmetic. In fixed-point arithmetic, numbers are represented with a fixed number of digits after the decimal point, ensuring consistent behaviour across platforms. [15], [16]

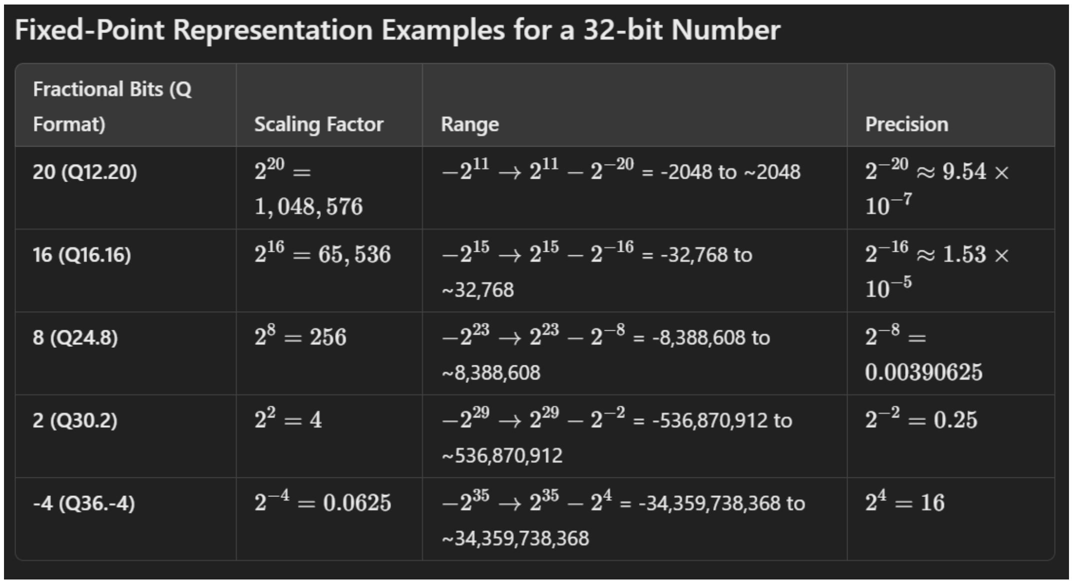
Using fixed-point numbers also has its disadvantages, primarily the limitations of range and precision in comparison to floating-point numbers. Fixed-point numbers represent a floating-point number, but they are stored as an integer. That means that if you want a wider range of values you will have to sacrifice precision and vice versa. [16]

In GPU-based fluid simulations, fixed-point arithmetic ensures deterministic outcomes regardless of hardware differences. This is because fixed-point numbers are essentially integers with limited range and precision, and integer arithmetic is guaranteed to have deterministic results within that range and precision.

The remainder of this section will explain the representation, type conversions, and operations of fixed-point arithmetic as described in [15], [17].

2.3.1. REPRESENTATION

A fixed-point representation of a fractional number is essentially an integer that is to be implicitly multiplied by a fixed scaling factor. The scaling factor could be anything based on the project’s needs. However, scaling factors are often chosen to be powers of 2 or 10. This representation allows standard integer arithmetic logic units to perform rational number calculations.



Fixed-Point Representation Examples for a 32-bit Number

Fractional Bits (Q Format)	Scaling Factor	Range	Precision
20 (Q12.20)	$2^{20} = 1,048,576$	$-2^{11} \to 2^{11} - 2^{-20}$ = -2048 to ~2048	$2^{-20} \approx 9.54 \times 10^{-7}$
16 (Q16.16)	$2^{16} = 65,536$	$-2^{15} \to 2^{15} - 2^{-16}$ = -32,768 to ~32,768	$2^{-16} \approx 1.53 \times 10^{-5}$
8 (Q24.8)	$2^{8} = 256$	$-2^{23} \to 2^{23} - 2^{-8}$ = -8,388,608 to ~8,388,608	$2^{-8} = 0.00390625$
2 (Q30.2)	$2^{2} = 4$	$-2^{29} \to 2^{29} - 2^{-2}$ = -536,870,912 to ~536,870,912	$2^{-2} = 0.25$
-4 (Q36.-4)	$2^{-4} = 0.0625$	$-2^{35} \to 2^{35} - 2^{4}$ = -34,359,738,368 to ~34,359,738,368	$2^{4} = 16$

Figure 11: Example of the range and precision of fixed-point numbers using different amounts of fractional bits (ChatGPT)

As shown in figure 11, a higher number of fractional bits results in a smaller range of values but offers greater precision in return. On the other hand, a lower number of fractional bits results in reduced precision but increases the range of values representable.

2.3.2. TYPE CONVERSIONS

To convert a number from floating point to fixed point, one may multiply it by the scaling factor S, then round the result to the nearest integer. Depending on the scaling factor and storage size, and on the input numbers, the conversion may not entail any rounding.

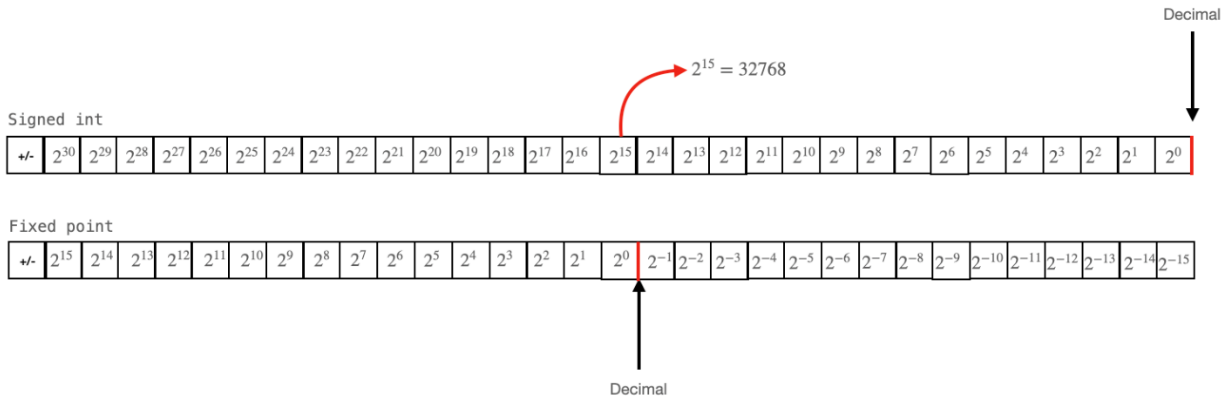
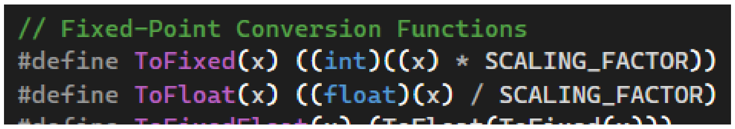


Figure 12: Example of the scaling factor calculation for a fixed-point with 15 fractional bits (Adams V. Hunter)

To convert a fixed-point number to floating-point, one may convert the integer to floating-point and then divide it by the scaling factor S. This conversion may entail rounding if the integer's absolute value is greater than 2^{24} (for binary single-precision IEEE floating point) or of 2^{53} (for double-precision). Overflow or underflow may occur if |S| is very large or very small, respectively.

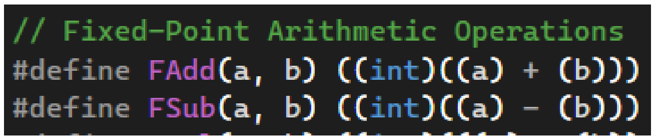


```
// Fixed-Point Conversion Functions
#define ToFixed(x) ((int)((x) * SCALING_FACTOR))
#define ToFloat(x) ((float)(x) / SCALING_FACTOR)
```

Figure 13: Type conversion macros between a floating-point and a fixed-point

2.3.3. OPERATIONS

To add or subtract two fixed-point values with the same scaling factor, it is sufficient to add or subtract those values. If the operands have different scaling factors, then they must be converted to a common scaling factor before the operation.



```
// Fixed-Point Arithmetic Operations
#define FAdd(a, b) ((int)((a) + (b)))
#define FSub(a, b) ((int)((a) - (b)))
```

Figure 14: Macros for fixed add and subtract operations on fixed values with the same scaling factor

For the multiplication or division of two fixed-point values, the risk for overflow or underflow to occur is much bigger. Let's look at multiplication to illustrate this problem. When we multiply two fixed-point values, each of which with 17 bits above the decimal (16 integer and 1 sign) and 15 bits below the decimal, then we end up with an intermediate value that has 2x17 = 34 bits above the decimal, and 2x15 = 30 bits below the decimal, for a total length of 64 bits. The only bits we are interested in are the bits around the decimal point, as shown in figure 15.

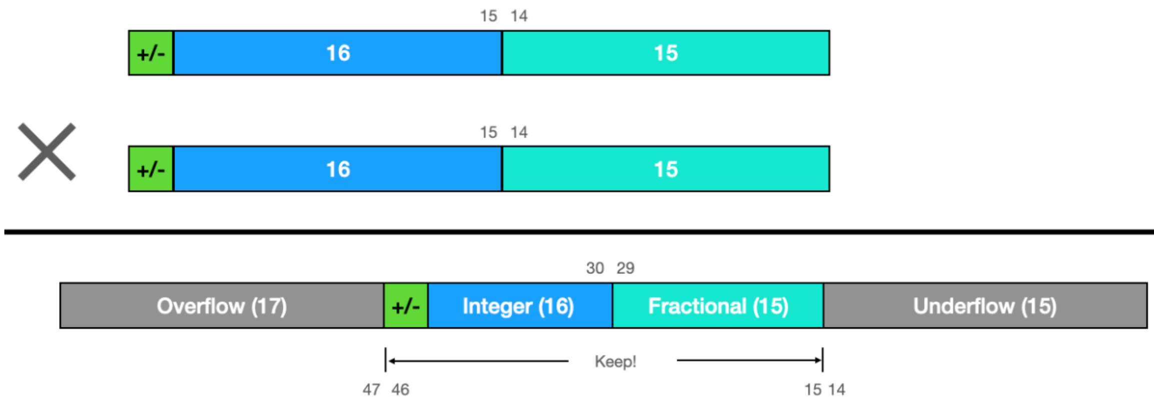
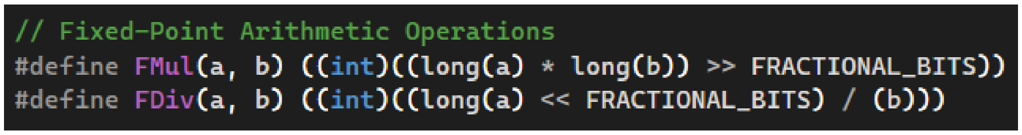


Figure 15: Example of a multiplication between two fixed-point values with 15 fractional bits (Adams V. Hunter)

To address this, we can typecast the operands to a higher precision and multiply those values. After that, we have to shift the result to the right and typecast the result back to the desired type. Division works a little bit different, because we have to shift the dividend to the left before dividing by the divisor, and then also typecast the result back to the desired type.



```
// Fixed-Point Arithmetic Operations
#define FMul(a, b) ((int)((long(a) * long(b)) >> FRACTIONAL_BITS))
#define FDiv(a, b) ((int)((long(a) << FRACTIONAL_BITS) / (b)))
```

Figure 16: Macros for fixed multiplication and division operations on fixed values with the same scaling factor

RESEARCH

1. INTRODUCTION

This research attempts to **implement fixed-point arithmetic in a GPU-based environment to ensure determinism**. The environment that I will be using during this research project is a fluid simulation because of how heavily they benefit from GPU-acceleration, making them an ideal choice for this project.

Because the focus of this research is on the implementation of fixed-point arithmetic, I will be using the code provided by [18] as a starting point for this research. This includes a fully working grid-based fluid simulation, based on the “Stable Fluids” algorithm as explained in [6], that runs on the GPU, and is made in Unity. There is also some functionality included to interact with the fluid simulation. It is worth noting that this implementation was last updated 7 years ago which explains why it doesn’t use, for example, Unity’s Input System Package [19] included by default in Unity 6.

I will not be making any unnecessary changes to this code base, and only work upon it by adding the features that are needed for this research. These features include a system to re-simulate the inputs from a previous simulation, and tools to measure performance and accuracy. More information on these measurements can be found in the next section of this Research chapter.

After executing the floating-point measurements, we will implement fixed-point arithmetic into the fluid simulation. Followed by, measuring the performance and accuracy of this fixed-point approach, and comparing the results to those from the floating-point version.

Listed below are the hard- and software specifications used to execute all the measurements for this research:

Hardware

- CPU: AMD Ryzen 7 5800H @ 3.20 GHz
- GPU: NVIDIA GeForce RTX 3060 Laptop GPU
- RAM: 16 GB @ 3200 MHz

Software

- Operating System: Windows 11, version 23H2
- Unity Engine: 6000.0.26f1
- Python: 3.13.1

2. MEASUREMENT METHODS

In general, all measurements apply a trimmed mean approach [20]. We will exclude 10% of the extremes on both ends of the spectrum before averaging the remaining 80% of results. For this research we will repeat every measurement 10 times, and thus only exclude the minimum and maximum result from those 10 iterations.

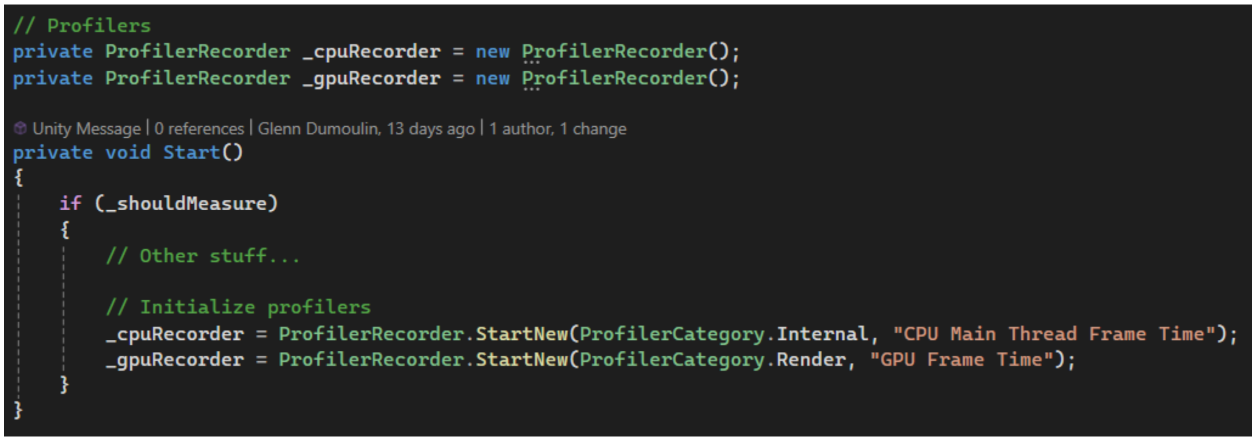
I also use JSON files to store the data before analysing the results using a Python script. Which in hindsight was not the best way to go about this, but there was not enough time left to improve the measurement approach and retake all measurements, and the used approach gets the job done.

The alternative approach for measuring accuracy would have revolved around encoding the fluid state to an image format, as shown in [21], to have much more accurate results, and as a bonus would it also require less disc space to save the results.

Next, we will take a look at the methods used for gathering the performance and accuracy data, followed by explaining how this data is then analysed.

2.1. PERFORMANCE

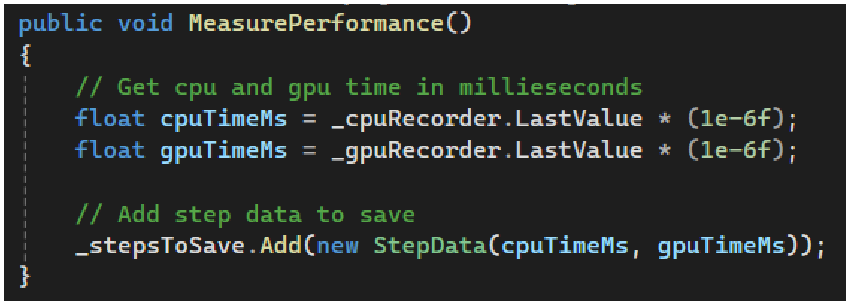
For gathering the performance data I used a couple of ProfilerRecorders from Unity’s Profiling tools [22]. One for the CPU Frame Time and one for the GPU Frame Time.



```
// Profilers
private ProfilerRecorder _cpuRecorder = new ProfilerRecorder();
private ProfilerRecorder _gpuRecorder = new ProfilerRecorder();

Unity Message | 0 references | Glenn Dumoulin, 13 days ago | 1 author, 1 change
private void Start()
{
    if (_shouldMeasure)
    {
        // Other stuff...

        // Initialize profilers
        _cpuRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Internal, "CPU Main Thread Frame Time");
        _gpuRecorder = ProfilerRecorder.StartNew(ProfilerCategory.Render, "GPU Frame Time");
    }
}
```

Figure 17: Code for initializing the CPU and GPU ProfilerRecorder to measure performance

After initializing the ProfilerRecorders, we get the latest value from each of them at the end of every fixed update. Note that these frame timing stats run a few frames behind so the first couple of frames will give invalid or unreliable data.



```
public void MeasurePerformance()
{
    // Get cpu and gpu time in millieseconds
    float cpuTimeMs = _cpuRecorder.LastValue * (1e-6f);
    float gpuTimeMs = _gpuRecorder.LastValue * (1e-6f);

    // Add step data to save
    _stepsToSave.Add(new StepData(cpuTimeMs, gpuTimeMs));
}
```

Figure 18: Code for getting the latest frame timing stats from the CPU and GPU ProfilerRecorder

When we want to analyse the performance measurements, we need to account for these invalid data entries. We will do this by once again adopting the trimmed mean approach. This way the result will also give a better interpretation of the actual average performance results, because we are excluding the timing stats for frames where either the latest value has not received its first value, or the CPU or GPU might have still been initializing.

2.2. ACCURACY

For gathering the accuracy data there was a bit more work to do. What these measurements try to validate is whether or not the simulation is deterministic, and the state of the fluid simulation is identical to those from other measurement iterations, given that the state is captured at the exact same time for each iteration.

We will run accuracy measurements after a certain number of seconds and will repeat this for several different timings. Our accuracy measurements will be taken after three different timings, those timings being 5 seconds, 10 seconds, and 20 seconds. The results from these different timings should indicate that different iterations deviate increasingly more from each other the longer the application is running, only if the simulation is not deterministic. However, if the simulation is deterministic, the accuracy results should reveal that there are no differences at all between iterations.

After the given timing has passed and the accuracy should be measured, the current state of the fluid simulation will be captured. This capture includes the velocity field and the colour buffer at that point in the simulation, and then we take a sample from this state capture. The sample we will use is grid-based, taking only the values from every 4th column of every 4th row. A visualization of this sampling method can be seen in figure 19 below.

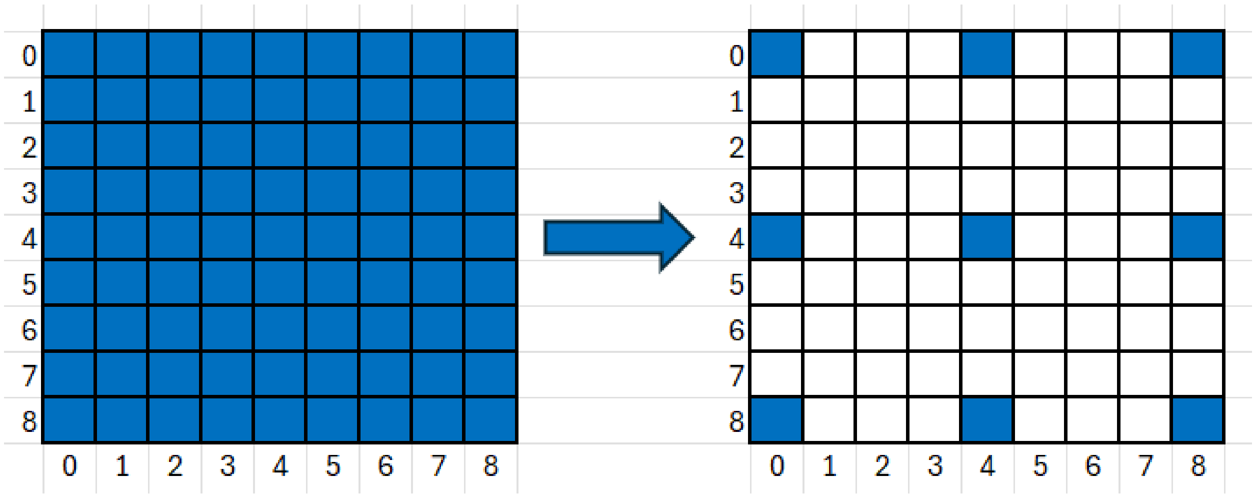
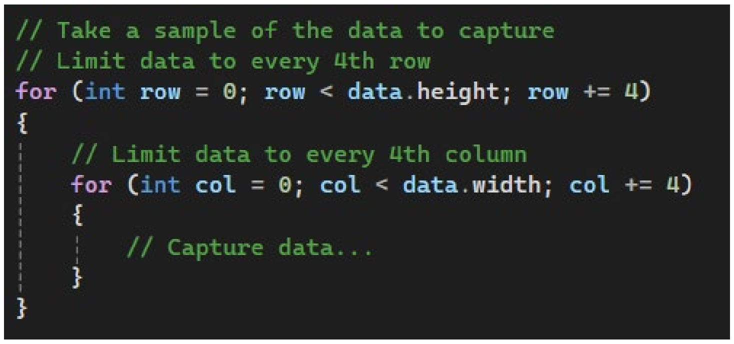


Figure 19: Example of capturing full state (left) vs capturing 4x4 grid-based state sample (right)



```
// Take a sample of the data to capture
// Limit data to every 4th row
for (int row = 0; row < data.height; row += 4)
{
    // Limit data to every 4th column
    for (int col = 0; col < data.width; col += 4)
    {
        // Capture data...
    }
}
```

Figure 20: Code for capturing 4x4 grid-based state sample

Looking back at this sampling method, I should have taken a bigger sample to measure the accuracy of the simulation more accurately, because the sample that was used leaves much room for non-deterministic results outside of the sampled area.

When we want to analyse the accuracy measurements, we calculate the differences between two iterations by comparing the values from the sampled velocity and colour data and repeat this difference calculation for each possible combination of two measurement iterations.

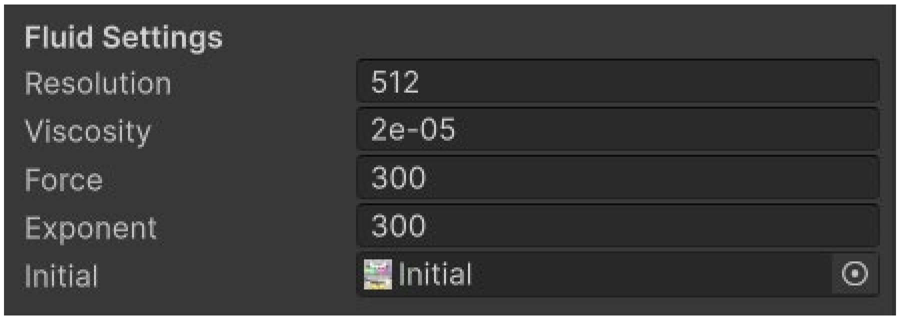
This means that for the accuracy measurements we will exclude more than only the minimum and maximum result from the trimmed mean, because from 10 iterations we already have 9 results comparing iteration 1 against all the other iterations. Then we compare iteration 2 against all iterations from iteration 3 onwards, since we already compared it against iteration 1 earlier, and this process gets repeated until all possible comparisons are executed.

3. FLOATING-POINT FLUID SIMULATION

This section will present the fluid parameters that were uses throughout the entirety of this research project. Followed by the implementation of a system to re-simulate the inputs from a previous simulation and closing out with the measurement results from the final version of the floating-point fluid simulation.

3.1. FLUID PARAMETERS

To ensure that the results from all measurements that will be performed throughout the entirety of this research project are directly comparable to each other, some common parameters have been chosen for the fluid simulation. The values for these parameters can be found in figure 21 below.



Fluid Settings

Resolution	512
Viscosity	2e-05
Force	300
Exponent	300
Initial	Initial

Figure 21: Common parameters for the fluid simulation

3.2. RE-SIMULATING INPUTS

The next step in the project will be to allow inputs from a simulation to be recorded and re-simulated at a later time. The reason why we want to have this functionality is to be able to execute measurement iterations that can be compared with each other. Primarily, this will be used for the accuracy measurements because we want to measure the state of the fluid simulation multiple times and validate whether the outcomes are deterministic or not. If not all iterations for this measurement have the exact same inputs, validating the determinism of the fluid simulation is not even an option.

It is also worth noting that in order to validate the determinism of the fluid simulation, we only care about the inputs that are made to interact with the simulation, and not about the outcomes from those inputs. The reason behind this is that we need to validate whether or not the calculations that happen to apply each input are deterministic, and simply applying the outcomes of an input holds no value for this research.

The first small, but important change that has to be made, is to make sure that the fluid simulation is updated at a constant rate. Luckily for us, Unity provides this functionality by default in their engine in the form of a fixed update. [23] This fixed update is called framerate independently at a fixed time step, which will be every 0.01 seconds or 100 times per seconds during this project.

Each time the fixed update is called we will increment a counter to keep track of during which step, what input should be executed. This step counter will then be used for both the saving and the loading of inputs.

To save an input to be re-simulated in later simulations, all we really need to do is know where the input is coming from, what direction it is going, and what kind of input it is. However, to ensure the re-simulation of this input happens during the same fixed update step in later simulations we will also need to store the current value of the step counter.

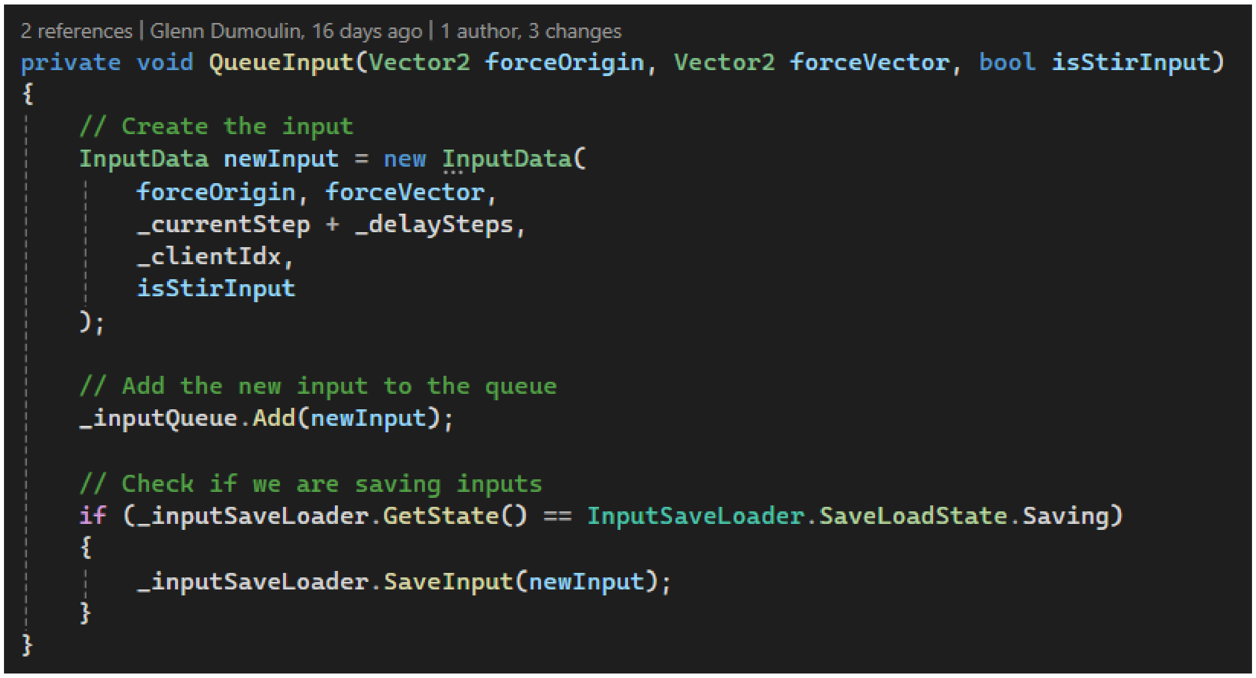
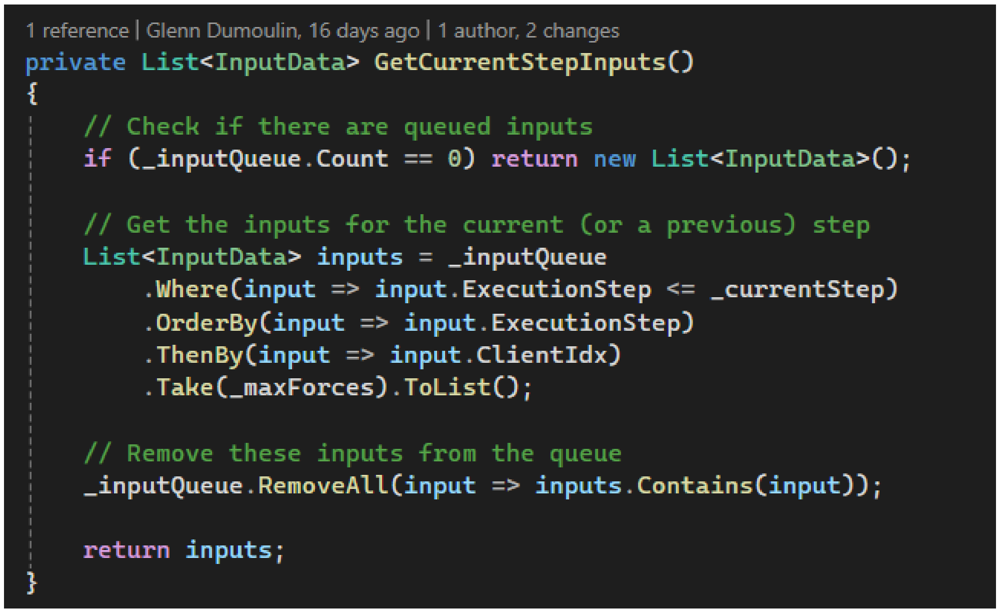

```
2 references | Glenn Dumoulin, 16 days ago | 1 author, 3 changes
private void QueueInput(Vector2 forceOrigin, Vector2 forceVector, bool isStirInput)
{
    // Create the input
    InputData newInput = new InputData(
        forceOrigin, forceVector,
        _currentStep + _delaySteps,
        _clientIdx,
        isStirInput
    );

    // Add the new input to the queue
    _inputQueue.Add(newInput);

    // Check if we are saving inputs
    if (_inputSaveLoader.GetState() == InputSaveLoader.SaveLoadState.Saving)
    {
        _inputSaveLoader.SaveInput(newInput);
    }
}
```

Figure 22: Code for saving an input to be re-simulated in later simulations

As mentioned in the Preface chapter of this paper, I originally wanted to synchronize this fluid simulation over a network, which is why there are still traces of preparations for the networking implementation remaining in the code base. In figure 22 above, for example, this is the inclusion of a number of delay steps to accommodate for networking delays, and the presence of a client index. This does mean that the inputs intentionally run a few frames behind, but this does not affect the measurements because this has always been the case throughout this research.

If we then for a later simulation want to re-simulate the inputs from a previous one, we can load the saved inputs back into the project and queue them up to be executed when the step counter of the fixed update reaches the saved input's execution step.



```
1 reference | Glenn Dumoulin, 16 days ago | 1 author, 2 changes
private List<InputData> GetCurrentStepInputs()
{
    // Check if there are queued inputs
    if (_inputQueue.Count == 0) return new List<InputData>();

    // Get the inputs for the current (or a previous) step
    List<InputData> inputs = _inputQueue
        .Where(input => input.ExecutionStep <= _currentStep)
        .OrderBy(input => input.ExecutionStep)
        .ThenBy(input => input.ClientIdx)
        .Take(_maxForces).ToList();

    // Remove these inputs from the queue
    _inputQueue.RemoveAll(input => inputs.Contains(input));

    return inputs;
}
```

Figure 23: Code for getting the inputs that need to be handled during the current fixed update

3.3. MEASUREMENT RESULTS

With the system to re-simulate inputs in place, the project is finally ready for the floating-point measurements. We will measure performance for two scenarios, once using loaded inputs that are re-simulated, and once using unique inputs for each iteration. For the accuracy we will handle the measurements as explained in the Measurement Methods section of this chapter, thus after 5 seconds, 10 seconds, and 20 seconds. All the accuracy measurements will logically use the same loaded inputs that are re-simulated.

3.3.1. PERFORMANCE

	Loaded Inputs	Unique Inputs
Average CPU Frame Time	0,46863ms	0,49938ms
Average GPU Frame Time	0,01910ms	0,01839ms

Table 1: Measurement results for the performance of the floating-point fluid simulation

3.3.2. ACCURACY

	After 5 seconds	After 10 seconds	After 20 seconds
Average Velocity Diff	0,00001	0,00009	0,00026
Average Colour Diff	0,00002	0,00032	0,00093

Table 2: Measurement results for the accuracy of the floating-point fluid simulation

3.3.3. CONCLUSION

These measurement results will mainly be used to compare against the measurement results we will get from the fixed-point fluid simulation. However, it is still worth mentioning a couple of observations.

When I first looked at the performance results, I actually thought something went wrong with the measurements because of how extremely short the frame timings were, with only half a millisecond for the CPU and only one fiftieth of a millisecond for the GPU. These values blew my mind, especially the GPU values, and I even tested this by giving the GPU a much big workload to check if the frame times would be longer that way.

From the accuracy results, I basically got what I expected to get. There are differences detected between measurement iterations and these differences only get bigger, the longer the project is running. This proves that floating-point arithmetic is not deterministic and will accumulate errors over time. As mentioned in the Measurement Methods section of this chapter, the sampling method used leaves much room for non-deterministic results outside of the sampled area, but even this very small sample seems to be enough in this case.

4. FIXED-POINT FLUID SIMULATION

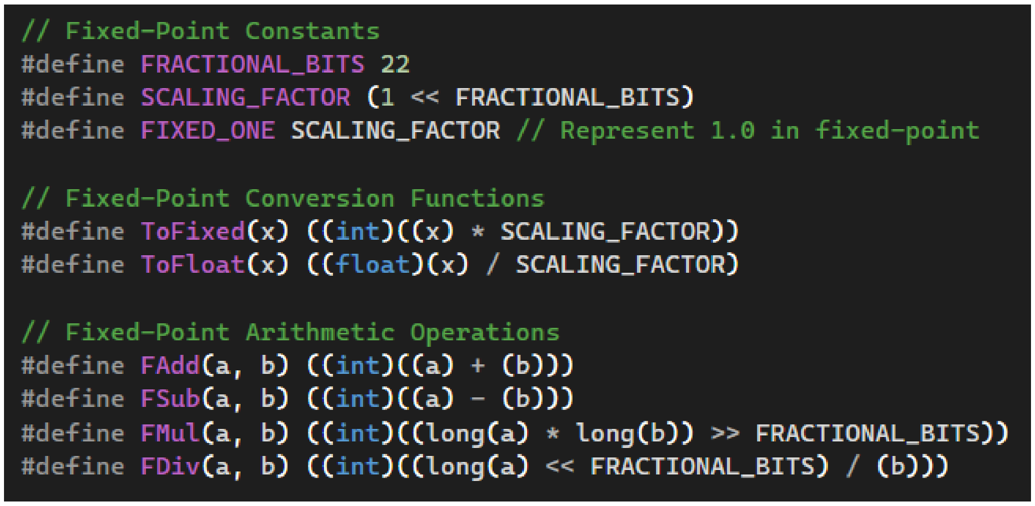
This section will present the implementation of fixed-point arithmetic. Followed by a workaround for fixed-point arithmetic in the form of fixed-point rounding and closing out with the measurement results from the final version of the fixed-point fluid simulation.

4.1. FIXED-POINT ARITHMETIC

Now that we have everything set up and the measurements for the floating-point fluid simulation have been handled, it is finally time to get started on the research part of this project, the implementation of fixed-point arithmetic. As explained in the Theoretical Framework chapter of this paper, fixed-point values are technically a simple integer value scaled by a certain scaling factor to represent a floating-point value.

The reason why we want to use fixed-point arithmetic is to ensure deterministic results for the fluid simulation, because, as proven by the floating-point accuracy results, floating-point arithmetic cannot provide that.

At first glance, the fixed-point arithmetic implementation seems pretty straightforward and fairly easy. Following the implementation which is shown and explained in more detail in the Theoretical Framework chapter of this paper, one would expect the implementation to look like what is shown in figure 24 below.

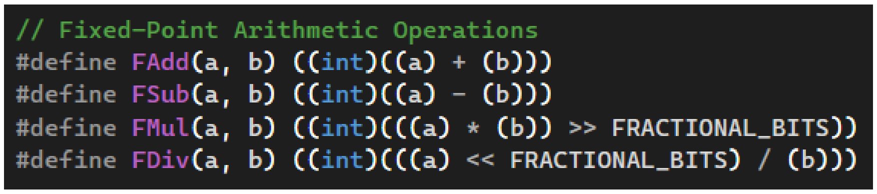


```
// Fixed-Point Constants
#define FRACTIONAL_BITS 22
#define SCALING_FACTOR (1 << FRACTIONAL_BITS)
#define FIXED_ONE SCALING_FACTOR // Represent 1.0 in fixed-point

// Fixed-Point Conversion Functions
#define ToFixed(x) ((int)((x) * SCALING_FACTOR))
#define ToFloat(x) ((float)(x) / SCALING_FACTOR)

// Fixed-Point Arithmetic Operations
#define FAdd(a, b) ((int)((a) + (b)))
#define FSub(a, b) ((int)((a) - (b)))
#define FMul(a, b) ((int)((long(a) * long(b)) >> FRACTIONAL_BITS))
#define FDiv(a, b) ((int)((long(a) << FRACTIONAL_BITS) / (b)))
```

Figure 24: Expected code for a fixed-point arithmetic implementation in HLSL

However, this is not the case. The code shown in figure 24 above will not even compile. This is because the long data type does not exist in HLSL, which is the programming language used for compute shaders and shaders in Unity. Only when using a shader model of 6.0 or higher there is an equivalent data type available called int64_t, as explained in [24]. This data type would most likely be what we need in this situation. However, I have been unsuccessful in finding a way to set the compute shader or shader to a target of shader model 6.0 in Unity.

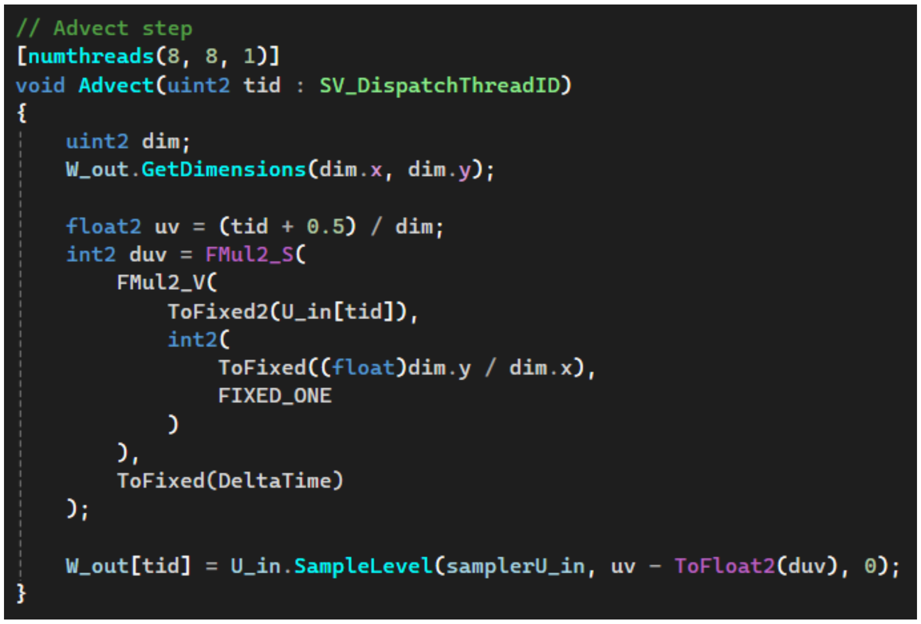
This leaves me with an implementation for the fixed-point arithmetic operations that looks like what is shown in figure 25 below.



```
// Fixed-Point Arithmetic Operations
#define FAdd(a, b) ((int)((a) + (b)))
#define FSub(a, b) ((int)((a) - (b)))
#define FMul(a, b) ((int)(((a) * (b)) >> FRACTIONAL_BITS))
#define FDiv(a, b) ((int)(((a) << FRACTIONAL_BITS) / (b)))
```

Figure 25: Actual code for fixed-point arithmetic operations in HLSL

And an example of the actual use of this approach in calculations can be seen in figure 26 below.



```
// Advect step
[numthreads(8, 8, 1)]
void Advect(uint2 tid : SV_DispatchThreadID)
{
    uint2 dim;
    W_out.GetDimensions(dim.x, dim.y);

    float2 uv = (tid + 0.5) / dim;
    int2 duv = FMul2_S(
        FMul2_V(
            ToFixed2(U_in[tid]),
            int2(
                ToFixed((float)dim.y / dim.x),
                FIXED_ONE
            )
        ),
        ToFixed(DeltaTime)
    );

    W_out[tid] = U_in.SampleLevel(samplerU_in, uv - ToFloat2(duv), 0);
}
```

Figure 26: Example of using the fixed-point arithmetic operations in HLSL

The issue here is that this code is prone to overflow because we are not accounting for it by typecasting to a higher precision before doing multiplication or division operations. Unsurprisingly, overflow has also been the main issue I have been having while attempting to implement fixed-point arithmetic.

Now, this is probably also the best moment to address why I am using a grand total of 22 fractional bits for my fixed-point values. Well, from my little bit of testing it seems like 22 fractional bits is the only amount that does not cause overflow for the larger values that occasionally occur throughout the simulation update, but using less fractional bits also offers too little precision. That means that inputs do not actually have any effect on the fluid simulation, rendering the fluid simulation static, no matter the inputs given.

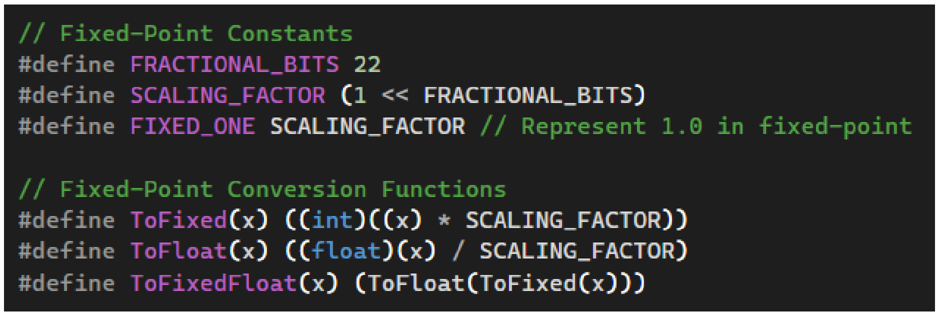
Let’s summarize what we have learned thus far. The fluid simulation requires a large number of fractional bits to register values from inputs that will not round down to zero before getting applied to the velocity field, for example. But the main issue we are facing is the inability to typecast the fixed-point values to a higher precision before doing multiplication or division operations, causing overflow for many operations.

4.2. FIXED-POINT ROUNDING

We attempted to implement fixed-point arithmetic, but that has failed. However, I still want to have some results for this research even if they are not perfectly deterministic, we might make a move in the right direction.

To try and achieve this, I decided to try something new that I did not expect to work because it honestly feels like it should not work. I am also guessing that the phrasing of that previous sentence might have given away that it did in fact work after all. Now, let’s take a look at what this approach actually is.

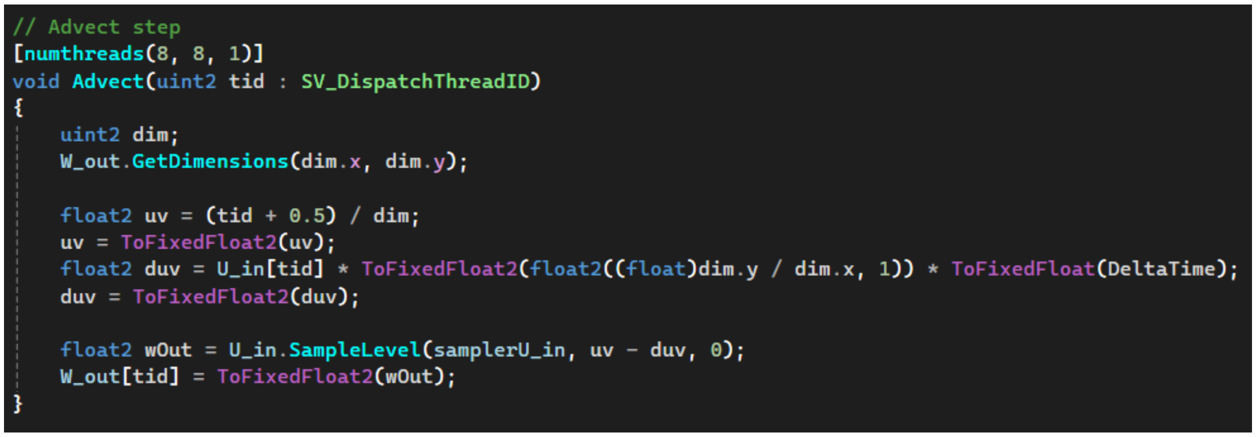
I call it fixed-point rounding, because what this approach basically only does is typecast intermediate values before and after operations to a fixed-point value and immediately typecast it back to a float. The code for this is then also remarkably simple and can be seen in figure 27 below.



```
// Fixed-Point Constants
#define FRACTIONAL_BITS 22
#define SCALING_FACTOR (1 << FRACTIONAL_BITS)
#define FIXED_ONE SCALING_FACTOR // Represent 1.0 in fixed-point

// Fixed-Point Conversion Functions
#define ToFixed(x) ((int)((x) * SCALING_FACTOR))
#define ToFloat(x) ((float)(x) / SCALING_FACTOR)
#define ToFixedFloat(x) (ToFloat(ToFixed(x)))
```

Figure 27: Code for implementing the fixed-point rounding approach in HLSL

And an example of the actual use of this approach in calculations can be seen in figure 28 below.



```
// Advect step
[numthreads(8, 8, 1)]
void Advect(uint2 tid : SV_DispatchThreadID)
{
    uint2 dim;
    W_out.GetDimensions(dim.x, dim.y);

    float2 uv = (tid + 0.5) / dim;
    uv = ToFixedFloat2(uv);
    float2 duv = U_in[tid] * ToFixedFloat2(float2((float)dim.y / dim.x, 1)) * ToFixedFloat(DeltaTime);
    duv = ToFixedFloat2(duv);

    float2 wOut = U_in.SampleLevel(samplerU_in, uv - duv, 0);
    W_out[tid] = ToFixedFloat2(wOut);
}
```

Figure 28: Example of using the fixed-point rounding approach in HLSL

Looking at this code, you might see why I did not expect this to work. All we are doing is rounding the operands of the operations based on the number of fractional bits we want and then typecast the rounded values back to floating-point values.

This entails that the operations themselves are still floating-point arithmetic and thus not guaranteed to give deterministic results. Even worse is, that this approach does not help with the issue of different GPUs implementing floating-point arithmetic differently in the slightest.

4.3. MEASUREMENT RESULTS

With the fixed-point rounding solution in place, the project is ready for the fixed-point measurements. We will measure performance for two scenarios, once using loaded inputs that are re-simulated, and once using unique inputs for each iteration. For the accuracy we will handle the measurements as explained in the Measurement Methods section of this chapter, thus after 5 seconds, 10 seconds, and 20 seconds. All the accuracy measurements will logically use the same loaded inputs that are re-simulated.

4.3.1. PERFORMANCE

	Loaded Inputs	Unique Inputs
Average CPU Frame Time	0,46821ms	0,50756ms
Average GPU Frame Time	0,01906ms	0,01831ms

Table 3: Measurement results for the performance of the fixed-point fluid simulation

4.3.2. ACCURACY

	After 5 seconds	After 10 seconds	After 20 seconds
Average Velocity Diff	0,00000	0,00000	0,00000
Average Colour Diff	0,00000	0,00000	0,00000

Table 4: Measurement results for the accuracy of the fixed-point fluid simulation

4.3.3. CONCLUSION

Let’s discuss a couple of observations from these measurement results and compare them against the measurement results from the floating-point fluid simulation.

When looking at the performance results, we can clearly see that the results are nearly identical to those from the floating-point measurements. From these results we can conclude that fixed-point rounding does not add any computational cost to the fluid simulation.

As stated previously when discussing the fixed-point rounding approach, the accuracy results received from these measurements are shocking to say the least. From these results it appears that the fluid simulation is now deterministic, even though we are still applying regular floating-point arithmetic. The only difference is that the floating-point values are rounded to a fixed number of fractional bits before and after any operation occurs.

Earlier in this section, we also acknowledged that the fixed-point rounding approach does not help with the issue of different GPUs implementing floating-point arithmetic differently. The outcome of this conclusion is that although the fluid simulation seems to be deterministic, we cannot guarantee that there will be deterministic results across different GPUs.

As mentioned in the Measurement Methods section of this chapter, the sampling method used leaves much room for non-deterministic results outside of the sampled area. In this case, using a larger sample would have given a more accurate result, since we would be checking a lot more values against each other.

DISCUSSION

While I expected that the performance would not really suffer from any significant impact due to the implementation of fixed-point arithmetic or fixed-point rounding, I did not expect the results to be so nearly identical to each other. As explained in the Theoretical Framework, fixed-point values are essentially an integer that is to be implicitly multiplied by a fixed scaling factor. Which led me to expect that the only minor increase in frame times would be due to the type conversions that happen between floating-point and fixed-point values.

As for the failed attempt to implement fixed-point arithmetic, I did expect the accuracy results to ensure deterministic results across different GPUs. The reason for this expectation is that integer arithmetic is inherently deterministic, for both the CPU and GPU.

On the other hand, and as mentioned before, I did not expect the accuracy results from the fixed-point rounding approach to result in deterministic outcomes. Perhaps using a larger sample for accuracy measurements would have given a more accurate result. However, due to limited resources and time, I could not verify whether or not this approach offers deterministic results across different GPUs. My expectation is that this will not be the case due to the non-standardized implementation of floating-point arithmetic in GPUs, resulting in deterministic outcomes per GPU but not when comparing the results from different GPUs to each other.

CONCLUSION

With this research project we tried to answer the question **“How does the implementation of fixed-point arithmetic impact the performance and accuracy of a fluid simulation in a GPU-accelerated environment?”**. The results from this research indicate that the use of fixed-point rounding offers a partial solution for deterministic fluid simulations in a GPU-based environment. This is only a partial solution because we are not actually using fixed-point arithmetic, but only applying a fixed-point rounding to the operands and outcomes of operations.

For the performance of the fluid simulation, there is no real impact, with frame times remaining nearly identical to those from floating-point measurements. From this we can conclude that fixed-point rounding does not add any computational cost to the fluid simulation.

For the accuracy of the fluid simulation, there is a significant impact, with fluid states from different measurement iterations remaining exactly equal to each other. From this we can conclude that fixed-point rounding does ensure deterministic results in a GPU-accelerated environment. However, further research is required to verify whether or not this is also the case when comparing these deterministic results with those from the fluid simulation on different GPUs.

There is also a need for further research to get a working implementation of the fixed-point arithmetic in a GPU-accelerated environment in Unity, in order to (dis)prove the guarantee of deterministic results across different GPUs.

FUTURE WORK

While this research project offers a partial solution for deterministic fluid simulation in a GPU-based environment, I firmly believe it can still become a lot better. By looking into ways to get the fixed-point arithmetic working as intended on the GPU, I expect the current partial solution to be improved to a fully working solution.

When a fully functional and deterministic solution has been implemented, my original research topic of synchronizing the fluid simulation over a network still holds value. The networking aspect brings its own difficulties and challenges to the table, such as handling latency and packet loss. These challenges already have certain ways of handling them, but to incorporate them into a project that is extremely sensitive to timing and precision will definitely offer an interesting research project.

This paper only focuses on 2D, but the “Stable Fluids” solvers work for 3D simulations as well. So, expanding the fluid simulation to 3D could be a first step for a new research project. After successfully expanding to 3D, the next steps could be many things, including adding objects such as boats or barrels and make them and the fluid interact with each other, or expand even further into an open-world environment.

Lastly, a bit of a crazier idea. I am not sure how possible this is at the time of writing this, but in the future, it may be very interesting to look into ways to guarantee deterministic behaviour across different platforms such as consoles and mobile devices. This can be very interesting and challenging to research because the fluid simulation heavily depends on the GPU to result in the best visuals the simulation can get. Most modern mobile devices do have some sort of GPU built in, but these are less powerful than those from computers or consoles.

CRITICAL REFLECTION

One of the biggest challenges for me has been balancing my ambition with the realistic scope of the project. I often overestimate what can be achieved within a limited period of time, which has led to moments of frustration. Thanks to my supervisor's continuous feedback, I was able to refine the scope of the project to something both feasible and meaningful. This has been a valuable lesson in how essential external perspectives are for staying grounded and focused.

This project has also taught me to not take on too much at once. This research topic was a case where I did not really take that into account, because basically all the topics included in this research were new to me. I did not have any prior experience with either fluid simulations or GPU programming, which resulted in having to learn about new techniques, methods, solutions, or workarounds for almost every task that I had to do.

Next to taking on too much, I also know that I often wait too long before asking for help if I am stuck on something. I want to prove to myself that I can in fact solve a certain problem, but usually this ends up taking an exceptionally long time. This has been the case for this project as well, where I only asked my supervisor for help when it was already extremely close to the final deadline. During that meeting with my supervisor, we discussed certain decisions I made along the way, and I learned the hard way that should I have asked for help sooner, both my project and results would have been a lot better.

I don't feel like I learned new things about myself during this project, but I was reminded of certain traits, such as my tendency to aim for perfection and overthink details. These traits drive me to strive for excellence, but they can also slow down progress. This project has reinforced the importance of finding a balance, recognizing that iterative progress is often more effective than waiting for the "perfect" solution.

One thing I did learn about myself during this project is that I struggle with reading long or dense texts. Early on, my supervisor recommended a book that I needed to read to build a foundation for my research. However, I found that when I tried to read it, I would quickly lose focus, get tired, or struggle to understand certain parts, which often required re-reading multiple times. This was frustrating and made me feel like I was making slower progress than I should.

To address this, I discovered a method that worked well for me: I found a PDF version of the book online and used Adobe Acrobat to have it read out loud, essentially turning it into an audiobook. Following along visually while listening helped me stay engaged, retain information better, and make the process much more enjoyable. This solution not only helped me complete the reading but also taught me the importance of finding learning methods that work for me.

This project has been a mix of pride and frustration. I have felt proud of the progress I have made in defining a unique research question and narrowing down the scope, but I have also been frustrated with myself for not progressing faster. These experiences have shown me that research is as much about persistence and adaptability as it is about knowledge and technical skill.

REFERENCES

[1] “Computational fluid dynamics,” *Wikipedia*. Aug. 19, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Computational_fluid_dynamics&oldid=1241121206

[2] “Navier–Stokes equations,” *Wikipedia*. Oct. 19, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Navier%E2%80%93Stokes_equations&oldid=1251968674

[3] R. A. Gingold and J. J. Monaghan, “Smoothed particle hydrodynamics: theory and application to non-spherical stars,” *Mon. Not. R. Astron. Soc.*, vol. 181, no. 3, pp. 375–389, Dec. 1977, doi: 10.1093/mnras/181.3.375.

[4] “Smoothed-particle hydrodynamics,” *Wikipedia*. Jul. 19, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Smoothed-particle_hydrodynamics&oldid=1235431985

[5] J. J. Monaghan, “Smoothed particle hydrodynamics,” *Rep. Prog. Phys.*, vol. 68, no. 8, p. 1703, 2005.

[6] J. Stam, “(PDF) Stable Fluids,” *ResearchGate*, Oct. 2024, doi: 10.1145/311535.311548.

[7] J. Stam, “Real-Time Fluid Dynamics for Games,” May 2003, [Online]. Available: https://www.researchgate.net/publication/2560062_Real-Time_Fluid_Dynamics_for_Games

[8] “Graphics processing unit,” *Wikipedia*. Dec. 04, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Graphics_processing_unit&oldid=1261206237

[9] “Unity Fluid Simulation Tutorial: CPU & GPU Methods.” [Online]. Available: https://daily.dev/blog/unity-fluid-simulation-tutorial-cpu-and-gpu-methods

[10] “Deterministic system,” *Wikipedia*. Jun. 13, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Deterministic_system&oldid=1092966327

[11] “Determinism :: Box2D.” [Online]. Available: https://box2d.org/posts/2024/08/determinism/

[12] T. B. Brown, “ML Tidbits: Nondeterminism on the GPU,” Tom B Brown. [Online]. Available: https://medium.com/tom-b-brown/tensorflow-nondeterminism-on-the-gpu-a0e86125fd06

[13] “Floating Point Determinism,” Gaffer On Games. [Online]. Available: https://gafferongames.com/post/floating_point_determinism/

[14] AlanZucconi, “The Further You Are From (0,0,0), The Messier Stuff Gets: Here’s How To Fix It ✨,” r/Unity3D. [Online]. Available: www.reddit.com/r/Unity3D/comments/ijwfec/the_further_you_are_from_000_the_messier_stuff/

[15] “Fixed-point arithmetic,” *Wikipedia*. Dec. 16, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Fixed-point_arithmetic&oldid=1263443554

[16] “Decoding Numerical Representation: Floating-Point vs. Fixed-Point Arithmetic in Computing,” DEV Community. [Online]. Available: https://dev.to/mochafreddo/decoding-numerical-representation-floating-point-vs-fixed-point-arithmetic-in-computing-3h46

[17] “FixedPoint.” [Online]. Available: https://vanhunteradams.com/FixedPoint/FixedPoint.html

[18] “keijiro/StableFluids: A straightforward GPU implementation of Jos Stam’s ‘Stable Fluids’ on Unity.” [Online]. Available: https://github.com/keijiro/StableFluids/tree/master

[19] “Input System | Input System | 1.11.2.” [Online]. Available: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.11/manual/index.html

[20] “Truncated mean,” *Wikipedia*. Jun. 26, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Truncated_mean&oldid=1162058593

[21] U. Technologies, “Unity - Scripting API: ImageConversion.EncodeToTGA.” [Online]. Available: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/ImageConversion.EncodeToTGA.html

[22] U. Technologies, “Unity - Scripting API: ProfilerRecorder.” [Online]. Available: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Unity.Profiling.ProfilerRecorder.html

[23] U. Technologies, “Unity - Scripting API: MonoBehaviour.FixedUpdate().” [Online]. Available: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/MonoBehaviour.FixedUpdate.html

[24] stevewhims, “Scalar data types - Win32 apps.” [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-scalar

ACKNOWLEDGEMENTS

I would like to thank some important people who have contributed to this paper.

I would like to start off by thanking Alex Vanden Abeele, my supervisor and Kasper Geeroms, my coach during this research project. They were always available to provide valuable feedback or simply lend a listening ear when I needed it.

I would also like to thank Keijiro for making the base project, which I used as a starting point for my research, and providing it through a GitHub repository.

Last but not least, I would like to give a special thanks to my parents who gave me the opportunity to study Digital Arts and Entertainment, and for the moral support in everything I have ever done.

APPENDICES

All the code that has been written to achieve this research project, handle the measurements and make the screenshots used in this paper has been made available online on GitHub. It also includes all results from measurements taken during the different stages of the project.

The repository is available at
https://github.com/GlennDumoulin/DeterministicFluidSimulation
and will be kept online as long as possible.