OpenGL GPU Features and SPH Fluid

Ewan Rice

August 22, 2016

Computer Animation and Visual Effects MSc

Bournemouth University

Abstract

This thesis covers the use of two General Purpose GPU features of OpenGL, Transform Feedback and Compute Shaders and the implementation of SPH fluid. Transform Feedback and Compute Shaders allow for arbitrary calculations to take place on GLSL shaders, making use of the parallelization of the GPU. An easy to use library for these features was written and an analysis and comparison of the two is made. The SPH fluid implementation closely follows the work of Clavet et al. and examines carrying out the simulation on the GPU.

Contents

1 Introduction

GPUs and the APIs designed to open them up to developers such as OpenGL and DirectX offer multi-threaded programming tailored for graphics which has been essential for real-time rendering. The uses for the GPU extend beyond real-time rendering and the video game industry, as computationally heavy research simulations, engineering and artistic graphics software all have a need for accelerated performance.

OpenGL is an API for rendering graphics, which is what the GPU is typically used for. However OpenGL also offers ways of using shaders to make any arbitrary computations. Some examples of areas in graphics that can benefit from this is to calculate the movement for particles, cloth and fluid or detect and resolve collisions. Transform Feedback and Compute Shaders are two methods that can be used for general purpose GPU programming. There are however other extensive APIs that offer the same in CUDA and OpenCL. These APIs offer more features however the need to install an external library and learn the procedures and languages associated with them can make Transform Feedback or Compute Shaders a simpler, and therefore preferable option [1].

A SPH fluid simulation offers a good learning opportunity for GPU programming.

2 Transform Feedback

Transform Feedback was introduced to OpenGL in version 3.0 and allows for writing to a buffer in a vertex shader, meaning any calculations made in the shader can be used after the fact and become accessible on the CPU [2]. Although it is perhaps out-dated by the introduction of Compute Shaders, it still has relevance for computers with graphics drivers that do not support version 4.3, where Compute Shaders were introduced, for example Mac machines [3].

Transform Feedback uses a vertex shader to perform the GPU-side calculations. A shader program must be created for the vertex shader to be attached to and compiled, as with a normal shader. The Transform Feedback is performed using two Vertex Buffer Objects, one as an input for sending data to the shader, and the other for writing the newly calculated data to an output from the shader. After the shader has finished, the data in these buffers must be swapped on the CPU, so in the next frame the input buffer now has the data that was output from the shader in the previous frame. This is sometimes referred to as ping-ponging.

2.1 Implementation

2.1.1 Create Shader Program

Firstly a shader program must be created which a vertex shader must be attached to, where the GPU calculations will take place. However, after attaching the shader and before linking the program a special call must be made in preparation for Transform Feedback.

```
glTransformFeedbackVaryings   (GLuint programID,
                               GLsizei count,
                               const char **varyings,
                               GLenum bufferMode)
```

[4]

This method is used to pass in the names of the varyings in the shader, which are the variables that are output from the shader.

programID	The ID of the program.
count	The number of varyings.
varyings	An array of strings of the names of the varyings, which must match the names in the shader.
bufferMode	Specifies whether a single buffer will be used for the varyings or multiple. It generally will be more convenient to use a single buffer. GL_INTERLEAVED_ATTRIBS for a single buffer, GL_SEPARATE_ATTRIBS for multiple.

It is important that varyings declared in the shader are explicitly written to and cannot be left unused, otherwise the shader won't successfully compile.

2.1.2 Set Data & Create VAO

The two VBOs must be generated and glBufferData called for each. Assign initialized data to what will be used as the input buffer, however the output buffer data may be set as a nullptr. The size of the output buffer must match the input buffer.

A Vertex Array Object can also be used for describing how the input attributes are to be sent to the shader. This can be created and reused for every frame the shader is used.

2.1.3 Executing Shader

1. Before executing the shader the following should be called -

glEnable(GL_RASTERIZER_DISCARD)

There is no need to draw or go further along the pipeline than this shader, which this call ensures.

2. Bind input buffer and VAO.

3. For each input attribute, call glEnableVertexAttribArray and glVertexAttribPointer. These attributes must align with the output varyings in terms of data types and order.

4. Bind output VBO using -

glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, m_outputVBO)

0 is the index of the buffer, if more buffers are used for the varyings, the index must match that.

5. glBeginTransformFeedback(GL_POINTS)

glDrawArrays(GL_POINTS,0,numIndices)

glEndTransformFeedback()

numIndices specifies the number of invocations of the shader. This may match the length of an array of the attributes that are being passed in.

6. Unbind the VBO and VAO.

7. glDisable(GL_RASTERIZER_DISCARD). Re-enable drawing post vertex shader.

8. Swap the input and output buffers std::swap(m_inputVBO, m_outputVBO).

Sample Vertex Shader

```
#version 330 core
layout (location=0) in vec4 position;
out vec4 outPosition;

void main(void)
{
    outPosition = position;
}
```

3 Compute Shader

Compute shader is another shader stage, the other stages being the Vertex, Fragment, Geometry and Tesselation shaders. A compute shader's purpose is to carry out computations on data, which may be used further in the pipeline for drawing or for any other heavy calculations [5].

- A significant difference between the compute shader stage and other shaders is that there are no-built in outputs from the shader, the user must send data to the shader using a buffer, texture or image and write out to one of these objects. This differs for example to the vertex shader in which the output gl_Position is built-in and must be written to.
- Uniform variables can still be used.
- Where a vertex shader is executed once per vertex that is passed into the shader, the number of times a compute shader is executed depends on user input.
- A compute shader is compiled and attached to a program in the same manner as any other shader type, with the difference being the GLenum GL_COMPUTE_SHADER is used when creating the shader.
- A compute shader cannot be attached to a program that other shaders are also attached to, it must be the only shader in a program [1][6].

3.1 Implementation

3.1.1 Work Groups & Dispatching Shader

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 [6].

Take an instance where we wish to carry out 64 threads of a compute shader, where 64 may be the length of an array of data which we wish to perform calculations with, or possibly the number of particles of which we wish to calculate the position. These 64 threads may be divided into 4 work groups of 16 threads (which we can also refer to work items). In reality, 64 threads can easily be run concurrently, but in a compute program that may be run millions of times, the number of concurrent threads needs to be divided down into more manageable groups. This is what work groups allows for and they can offer more control in the shader [6].

It is important that the correct number of work groups are used. In a case in which there are 102 particles, we may divide this into work groups of 16

work items(or threads). The number of work groups must be a positive whole number, and we might calculate this by the following-

totalParticles = 102

invocationsPerGroup = 16

numWorkGroups = 102 / 16 = 6.375

6.375 rounded to nearest lowest int is 6. If we use 6 work groups of 16 threads each, then a total of 6 * 16 = 96 threads are used. If 96 threads are run for 102 particles, this means 6 particles are not being considered in the compute shader. It is therefore necessary to increment numWorkGroups by 1, or make sure totalParticles is a multiple of invocationsPerGroup.

It is also up to the user to set the most efficient number of work groups and threads/invocations. More threads may improve performance though there will be a limit on the number of threads per work group. The maximum number of work groups can be queried with the OpenGL api using

glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT).

Similarly the maximum number of invocations may be queried with

glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS).

Other relevant variables are also available to query. [5]

3.1.2 Dispatching Shader & Work groups in 3D

To execute a shader from the CPU side a glUseProgram call is made as with any other type of shader, however instead of glDrawArrays as with a normal program, a call to glDispatchCompute is made.

glUseProgram(GLuint progamID)

glDispatchCompute(GLuint num_groups_x, GLuint numGroups_y, GLuint num_groups_z)

The space the groups work in is three dimensional [5]. This dispatch method takes in the number of work groups we wish to dispatch in the x, y and z. In practice, how many groups are dispatched in each direction depends on how we wish to use the compute shader. If the shader is being used to calculate the positions of particles, it is being executed once per particle, which are stored in a linear array. It is therefore easiest to dispatch the total necessary number of work groups in the x, and 1 work group in y and z. An example of dispatching 2 dimensional groups may be when writing to a texture or image, which can allow for more convenient access of that object [1].

3.1.3 Built-In Inputs

Compute shaders have the following built-in inputs [5].

uvec3 gl_NumWorkGroups	Number of work groups, same as what was passed in glDispatchCompute.
uvec3 gl_WorkGroupID	The current work group for a shader invocation.
uvec3 gl_LocalInvocationID	The current invocation of the shader within the work group. May be thought of as the ID of the current thread within in the current work group.
uvec3 gl_GlobalInvocationID	The unique ID of the current invocation across all invocations.
gluint gl_LocalInvocationIndex	The unique ID of an invocation within a work group.

In terms of a particle system on a compute shader program, if there are 128 particles and we wish to use 16 threads per group, which will give 8 work groups, the dispatch call would be -

glDispatchCompute(128 / 16, 1, 1)

For any one invocation of the shader

gl_NumWorkGroups = (8, 1, 1)

gl_WorkGroupID = (range(0,7), 0, 0)

gl_LocalInvocationID = (range(0,15), 0, 0)

gl_GlobalInvocationID.x = (range(0,127), 0, 0)

gl_LocalInvocationIndex = range(0,15)

Any of these inputs may be used in getting an index of an element of a SSBO.

3.2 Shader Storage Buffer Objects

As part of OpenGL 4.3 SSBOs were introduced as a new buffer object which unlike a Uniform Buffer Object can be both read and written to. This, along with a much larger maximum storage size make it very useful in combination with compute shaders. [7].

Like other buffers they must be generated, given data and bound to a location in the shader before each dispatch call.

They are defined in shaders as follows -

```
layout (std430, binding=0) buffer SSBO
{
    float buffer[ ];
};
```

The binding must match the location it is bound to on the CPU, and it can be defined without an array size, though the size will be determined by the memory allocated for the buffer on the CPU when the buffer was assigned data. The length of the buffer can be returned using .length( ).

3.2.1 std140 & std430

A potential source of problems can come from defining the layout as std140 or std430. A std140 layout does not guarantee that the buffer will be tightly packed. A buffer of floats will in fact have a stride of vec4, which if unexpected can produce strange results[8].

It can therefore be safer to define any buffers with std140 as of type vec4, or use std430 which has optimizations that ensure the tight packing of types other than vec4. In general it is best to avoid using vec3 all together.

3.2.2 Accessing the Buffer

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.

Sample Compute Shader

```
#version 430 core
#extension GL_ARB_compute_shader: enable
#extension GL_ARB_shader_storage_buffer_object: enable
#extension GL_ARB_compute_variable_group_size: enable

layout (std140, binding=4) buffer Pos
{
    vec4 positions[ ];
};

layout (local_size_x = 256, local_size_y = 1, local_size_z = 1) in;

void main()
{
    uint index = gl_GlobalInvocationID.x;
    positions[index].y += 0.1f;
}
```

4 GPU Particle System

A particle system can be implemented with Transform Feedback or Compute Shaders with very similar code and considerations. This simple implementation creates a particle system that does not require any of the particles to have any influence on each other, as each moves separate to each other, only dependant on it's own velocity and position, which makes it well suited to the parallelization of the GPU.

Particle effects generally consist of thousands of particles and can be used to create non-polygonal or deformable bodies such as rain, fire, water, smoke, magic effects, trail effects etc. Reeves produced an early particle system for Star Trek II: Wrath of Khan which introduced particles and the common attributes that are still often seen in modern systems [9].

In this implementation each particle is represented by a position, velocity and lifetime. The position of a particle is what is used in drawing a point on screen, however the velocity is used in calculating the position of the particle every frame, and the lifetime is stored to constantly re-use a particle, and create a steady stream.

Both the Transform Feedback and Compute Shader implementation use the same data types in the shader code -

Vec4 position

Vec4 initialPosition

Vec4 velocity

Vec2 lifetime

lifetime is represented as a vec2 as the x component stores the current lifetime of a particle, and the y stores the initial lifetime. When the current lifetime drops below 0, the x component is set to that initial value stored in the y component, velocity is set to 0, and position is set to the initial position. An alternative to storing the initial position would be to get a new random value whenever the particle is reset.

Differences lie in how the data is sent to the shaders -

Transform Feedback

A struct is made called Particle, which holds the variables of the above data types. A total number of particles is defined, and a single VBO is filled with that number of Particle objects that have been set with initial values. Calls to glVertexAttribPointer with the correct stride and location values need to be made in order to correctly pass the data to the vertex shader that will carry out the calculations. The output VBO is then swapped with the input VBO.

Compute Shader

To send the data to the compute shader, separate SSBOs are made for position, initialPosition, velocity, and lifetime. The SSBOs must be bound to their location in the shader code before dispatching the shader.

4.1 Particle Update

The particle update code is to be executed every frame, for each particle. It performs the following functions -

- Calculate the new position of the particle through Euler integration, using the velocity and the external force of gravity.
- Collision detection with any spheres that have been defined, and resolve those collisions.
- Check if the current lifetime has dropped below 0, if so, reset the particle.

Pseudocode

```
velocity += gravity * timeStep
```

```
for each Sphere s in spheres :
    velocity = collisionWithSphere(position, velocity, s.position, s.radius)
position += velocity * timeStep
currentLifetime -= timeStep
if currentLifetime < 0 :
    currentLifetime = initialLifetime
    position = initialPosition
    velocity = vec3(0,0,0) //May alternatively wish to set an initial velocity
```

4.2 Particle Collision with Sphere

A point colliding with a sphere can be calculated in GLSL using the reflect() method [1], which takes in a vector, and returns a reflection of that vector about an incident vector, which in this case will be the collision normal.

Pseudocode

```
Vec3 sphereCollision(position, velocity, s.position, s.radius)
    //Check if the distance from the particle to the
    //sphere is less than the sphere's radius
    if distance(position - s.position) < s.radius :
        collisionNormal = (position - s.position).normal
        //Bounce coefficient is a value between 0 and 1
        //which acts as a dampener. 1 will produce a perfect
        //bounce where as 0 produces no bounce at all.
        return reflect(velocity, collisionNormal) * bounceCoefficient
    else :
        return velocity
```

4.3 Drawing the Particles

The data that is on the VBO in a Transform Feedback implementation or in the SSBO storing positions can be passed directly to a vertex shader to be drawn, without the need to write to a new VBO, they must both just be bound before the call to glVertexAttribPointer is made using glBindBuffer(GL_ARRAY_BUFFER, bufferID).

5 OpenGL GPU Library

As part of this project, a library was written that would make it easier to use the Transform Feedback and Compute Shader features of OpenGL. The library provides wrapper classes for both of the methods as well as a general shader management class and SSBO wrapper class. The aims of the library would be to enable users to easily make use of the OpenGL functions related to the GPU features by reducing the number of method calls required and offering more descriptive method names which will help to avoid common mistakes and be a bit more accessible. However, the user still must write the GLSL shader code from scratch and will require some surface knowledge of how shaders are created and used on the CPU side, particularly the order certain functions are called.

There are 4 classes in the library, though depending on the program, a user may only need to concern themselves with TransformFeedback or ComputeShader.

ShaderManager

A general purpose class for creating and storing shader programs. It is used in other classes in the library, but may be used by the user to create other shader types other than those used for general purpose GPU features.

TransformFeedback

This class creates the shader program to be used for the Transform Feedback, and handles all other code related to executing the shader and making use of Transform Feedback. For every shader the user wishes to perform calculations on, an instance of this class should be created.

SSBO

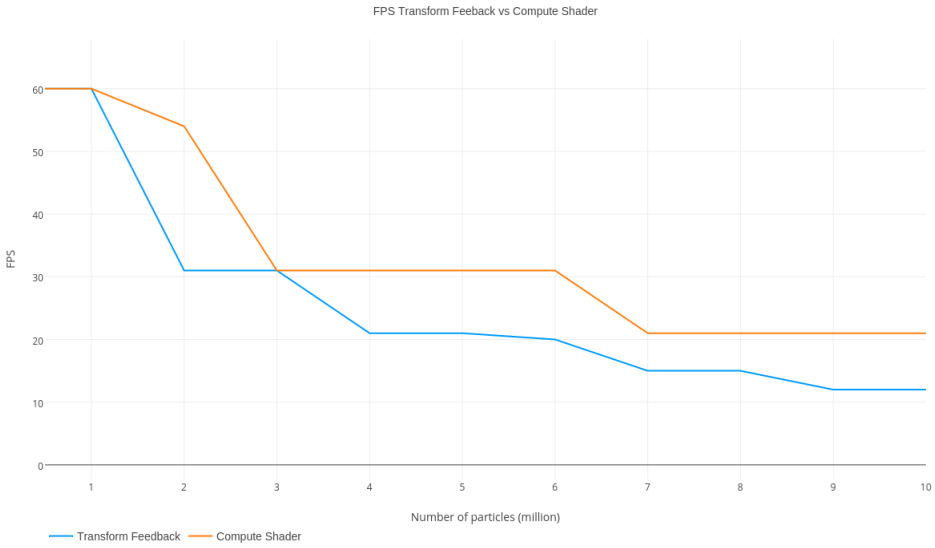
Generates a SSBO and stores the id. The class also offers methods for setting the data in the buffer and for binding. The ComputeShader class has methods that will create and store SSBO objects though users may want to create SSBO objects separately and use them where appropriate. For example if a program requires the same SSBOs to be used across multiple Compute Shaders, it may be easier and more logical for a user to create and store SSBO objects, rather than use one of those ComputeShader objects to create them all.

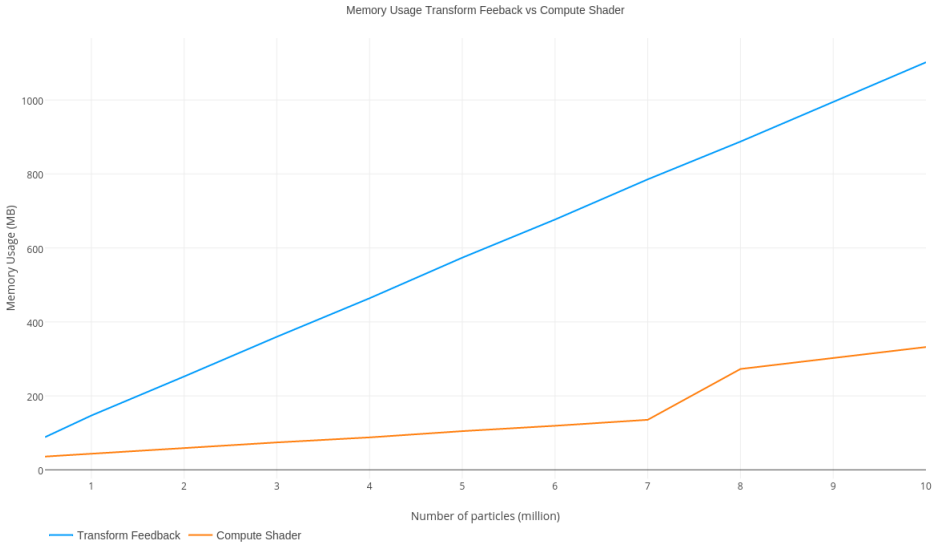
ComputeShader

Creates the compute shader program and attaches the shader. To dispatch the compute shader, users call useProgram() and then dispatch(). If any uniform variables are to to be set it must be done between the use and dispatch calls. This class can also create and store SSBOs for use.

6 Analysis and Comparison of Compute Shaders and Transform Feedback

Using the particle system from section 4, a test was run using both methods, examining the memory usage and FPS of the same simulation run with different numbers of particles. The shader code run in both methods involves the same operations for updating the data passed in, and keeps the same data types. The simulation involved particles falling under gravityand being re-used, and potentially colliding with 6 spheres. Below are the results from the tests -





6.1 FPS

Both programs targeted 60 FPS, and FPS is recoreded as the average FPS of a running program. The following observations are made -

- The results showed a higher and more consistent FPS for the Compute Shader particles.
- There is no linear decline in FPS with the increase in particles for either method.

6.2 Memory Usage

- Transform Feedback has a significantly greater memory usage than Compute Shaders. By taking the difference in memory at 2 and 3 million particles we find that Transform Feedback uses 107.4MB of memory for 1 million particles, where as Compute Shader uses 15.2MB. This can at least partly be attributed to the need for two VBOs being allocated in Transform Feedback as opposed to using SSBOs for Compute Shaders that can be both read and written to.
- There is a linear increase in memory for Transform Feedback with the number of particles used, however from 7 to 8 million particles, the memory usage doubles in the compute Shader program.

7 SPH Fluid Simulation and GPU Implementation

Smooth Particle Hydrodynamics is a fluid implementation first developed by Monaghan [10]. It is a particle-based method, putting it in the Lagrangian category of fluid simulations.

Background

Lagrangian Methods

Lagrangian fluid sims use particles to represent the fluid, where each particle may be thought of as a fluid molecule, and for which its velocity must be calculated to behave like fluid when surrounded with other particles. The velocity is then used to calculate the position of the particle for it to be drawn. An advantage of a particle-based method is that drawing the particles is relatively simple, though to represent the fluid as a body with a surface, rather than numerous single points in space, additional rendering methods such as marching cubes / meta-balls may need to be employed.

Creating a convincing fluid simulation requires a vast number of particles to be used. This can be a distinct disadvantage of particle-based methods as all fluid calculations are carried out per particle, therefore the complexity of the computations increases dramatically with the increase in the number of particles, with potentially very slow performance as a result.

The first implementation of SPH by Monaghan et al. from 1992, was used as part of research for astrophysics. Muller et al.[11] then presented SPH with graphical applications in 2003. However, the implementation that is discussed in detail below is based on Clavet et al. which introduced some improvements in the incompressibility aspects of SPH which resolves issues with particles clustering together, and changed the force based movement of particles to position based. Clavet et al. [12] also introduces springs between neighbouring particles as a way of handling elasticity, and covers fluid stickiness with other objects.

Euler Methods

Alternatively there is the Euler methods, which instead of particles, uses a grid that stores the velocities of a fluid in cells of the grid. Early work by Foster & Metaxas [13] was followed by Jos Stam [14], who presented a stable real-time implementation.

Hybrid Methods

Some hybrid implementations of Lagrangian and Euler such as the Affine PIC method by Jiang et al. calculate the fluid velocity on a grid which is then applied to particles placed in this grid [15]. This can be more efficient as a large number of particles can be used with a much less substantial drop in performance, as complexity is mostly determined by the resolution of the grid, where each cell has fluid calculations performed.

Navier Stokes

All the above methods have a basis in the Navier-Stokes equation, which describes the movement of a fluid with the following -

$\rho(\frac{\partial}{\partial t} + u \bullet \nabla)\mathbf{u}$ = -∇p + $\mu \nabla \bullet (\nabla \mathbf{u})$ +$\mathbf{f}$

$\nabla \bullet \mathbf{u} = \mathbf{0}$

Where ρ is density, p is pressure, u is velocity, μ is viscosity and f is the sum of any external forces such as gravity [16]. The first line is known as the momentum equation which describes the movement of fluid based on pressure, density, viscosity and external forces, all of which are factored into the SPH implementation by Clavet et al. The second line is the incompressibility condition which states that the mass and overall velocity of the fluid must remain constant [16]. With a particle-based approach this is achieved so long as the number of particles in the fluid remains constant [11].

7.1 SPH

In carrying out various fluid calculations the goal is to calculate a position for a particle every frame that will produce fluid like behaviour when viewed with thousands of other particles. This implementation of SPH closely follows Clavet et al. though leaves out the spring calculations while still achieving convincing fluid movement. Below are the steps that are carried out every frame of the program, in the order they are carried out.

1. for each particle:
   Apply external forces
2. for each particle:
   Apply viscosity
3. For each particle:
   Advect the particles
4. For each particle:
   Find all neighbouring particles
5.For each particle:
   Calculate density
   Calculate pressure
   Displace position of the particle and each neighbour
7. for each particle:
   Update velocity

for each particle:
   Resolve collisions

7.1.1 Kernel Function & Kernel Length

A number of constants are used that will be covered in the sections below however there is a common kernel function that is used in multiple calculations where a particle and its neighbour are being considered.

a = 1 - (d / k) [12]

Where d is is the distance between the particle and its neighbour, and k is the kernel length, a constant. Kernel length is the radius within which a particle will be affected by another particle. The function above is a measure of how close one particle is to it's neighbour, and should always be a value between 0 and 1. If a value less than 1 is returned, this means the neighbour is further from the particle than the kernel length, and so neither the particle nor neighbour should have an effect on each other.

7.1.2 Neighbour Searching

Particles must know which other particles are within a distance <= kernel length. It would be possible to do the following check -

Pseudocode

```
clear p1.neighbours

for each Particle p1 in particleList:

    for each Particle p2 in particleList:
```

```
    if (distance <= k)
        Add p2 to p1.neighbours
```

However in a simulation with thousands to hundreds of thousands of particles, this is much too inefficient to be usable.

There are two neighbour search methods that have been implemented in this project, both of which use a grid of cubic cells that all have a width, height and breadth of k, and are all equally spaced apart. The general idea is to know which particles are in which cell so the above algorithm may be written as -

Pseudocode

```
for each Cell c:
    for each Particle p1 in c:
        for each Particle p2 in cells neighbouring and including c:
            if (distance <= k)
                Add p2 to p1.neighbours
```

Therefore instead of comparing every particle against every other particle we are comparing only against the particles that are within a distance of, at the very most, 2 * k.

The disadvantage of using a grid to find neighbours is that the particles stay within that grid, which limits the space the simulation can take place in. Although the size of the grid may be increased this will have a cost in memory. An alternative solution called Spatial Hashing will be discussed below. An octree method and a simple grid method have been implemented.

Octree

An octree divides a space into smaller, equally sized divisions, and uses a tree data structure to find what one of those divisions a given point is in. For example given a point in space, within the bounds of the octree, firstly check which quadrant that point is located in, then find the quadrant of that quadrant, and so on for however many layers that we wish to check.

On initialization, all grid cells are added to the octree, with pointers to them stored within the smallest divisions of the tree. Every frame, particles are passed through the tree, until they are in the lowest division. This division will have cells in it, which a particle will be checked against and added to the cell if it is found to be located within its bounds.

Simple Grid

The following method was used as part of the GPU implementation as the tree data structure was less suitable within the limits of GLSL programming.

If we imagine a 3D grid with cells of width and height 1.0, where the first cell at index (0,0,0) in the grid has a left boundary at 0.0, and right boundary of 1.0. Similarly the bottom boundary is at 0.0, and top boundary at 1.0, and the same for the back and forward boundary. A point at position (2.13, 3.56, 9.90) will be in cell index (2,3,9).

For a 3D grid that has cells of arbitrary size and for which the cell at (0,0,0) does not have a lower, left, back boundary at (0.0, 0.0, 0.0), we must effectively convert the grid and the point we are looking at to the above size.

Firstly calculate the offset to move the grid to have it's lower, left, back at (0.0, 0.0, 0.0).

Pseudocode

x index = roundDownToInt((point.x + offset.x) / cellSize)

y index = roundDownToInt((point.y + offset.y) / cellSize)

z index = roundDownToInt((point.z + offset.z) / cellSize)

With the cell index, we can add this particle to the appropriate cell.

Spatial Hashing

An alternative solution, which bears some similarities to the Simple Grid method above is Spatial Hashing, which does not have the disadvantage of being confined to a grid. The world space is essentially divided into equally spaced cells, however the cells are not stored in memory as with the above solution. Every cell will have a unique hash key, and so particles that are within the same cell will have the same key. Storing particles by the hash key is a convenient way of finding neighbours. Every frame, the hash table that stores all the keys, and particles with those key, is cleared. For each particle, the hash key for the position it is in is found, if the key is already in the hash table, the particle is added to the list of particles stored with that key, otherwise the key is added to the table and the particle with it [17].

7.1.3 Density

Every particle has a density attribute, and it depends on the number of particles neighbouring it, and how close those particles are. The following formula is used to calculate density [12] -

$\rho = \sum_{j \in N(i)} (1 - rij/h)^2$

The square of the function from section 7.1.1 is used here. The formula states that the density of a particle is the sum of a^2 for all neighbouring particles.

Near Density

An important part of the solution by Clavet et al. is to introduce an additional density calculation which is called 'Near Density'. It is calculated similarly to the density but with a slight different kernel function of a^3.

Pseudocode

```
for each Particle p :
    p.density = 0
    p.nearDensity = 0
    for each Particle np in neighbours of p :
        p.density += (1 − (dist/k))^2
        p.nearDensity += (1 − (dist/k))^3
```

The introduction of near density was to solve the problem of particles clustering, and keeps particles spaced apart, as the molecules of a fluid should be.

7.1.4 Pressure

Pressure and Near Pressure is calculated for each particle using the particle's density and near density values.

Pseudocode

```
p.pressure = (p.density - restDensity) * stiffness
p.nearPressure = p.nearDensity * nearStiffness
```

restDensity is a constant that can control how closely particles may sit beside each other at rest. Higher restDensity will produce a much more dense fluid. If the particle density is greater than restDensity it will have a repulsive movement from other particles, if it is less it will have an attractive movement. stiffness is another constant, it is useful in controlling the pressure of particles, a higher stiffness will produce more wild and splashy particle movement.

The calculation for near pressure does not use restDensity. This means it will always contribute to a repulsive movement, which aids in keeping particles from clustering. nearStiffness is another constant which can give more control over the look of the simulation similar to the stiffness constant. It should generally be kept around the same value as stiffness, but it should be kept in mind that higher nearStiffness will only produce higher repulsive movement.

7.1.5 Displacement

Using the newly calculated pressure and near pressure values, particles must be moved, depending on those values, towards or away from neighbouring particles. For each neighbour, the particle is displaced, and the neighbour is displaced in the opposite direction.

Pseudocode

```
for each Particle p :
    Vec3 pDisplace
    for each Particle np in neighbours of p :
        v = np.position - p.position
        a = 1 - (v.length / k)
        displacement = timeStep^2 * (p.pressure * a + p.nearPressure * a^2) * v.normal
        np.position += displacment * 0.5
        pDisplace -= displacement * 0.5
    p.position += pDisplace
```

7.1.6 Viscosity

Viscosity is a measure of the thickness of a liquid, for example oil has a high viscosity. In terms of SPH the application of viscosity may be omitted and still produce fluid like behaviour, however with viscosity the simulation can be used to produce various types of fluid, and generally gives more control over the look of the simulation.

The viscosity calculations are carried out for the particle and each of its neighbours.

Pseudocode

```
for each Particle p :
    for each Particle np in neighbours of p :
        v = np.position - p.position
        u = dot( (p.velocity - np.velocity), v.normal )
        if (u > 0.0) :
            a = 1 - (v.length / k)
            impulse = timeStep * a * (linearVis * u + quadraticVis * u * u) * v.normal
            p.velocity -= impulse * 0.5
            np.velocity += impulse * 0.5
```

The variable u stores the projection of the difference in velocities of the two particles onto the vector between them. Checking if $u > 0$ means we are only applying viscosity to particles that are on a collision course. The constants linearVis and quadraticVis control the level of viscosity. For more viscous fluids, increase linearVis. For less viscous fluids linearVis may be reduced but quadraticVis should still remain > 0, as it is used in reducing velocity inside the fluid, away from the surface [12]. Without viscosity applied a potential unwanted behaviour is for the fluid to be constantly swirling without coming to a rest.

7.1.7 Advection

Particle positions are moved along based on their velocity. The particle's previous position is also stored before the advection, which will be used in updating the velocity in a later step.

Pseudocode

```
p.previousPosition = p.position
p.position += p.velocity * timeStep
```

7.1.8 Updating Velocity

Using the previous position and current position the velocity is updated. This is necessary as the velocity is not the only factor in moving the particle. The displacement step will move a particle's position not based on velocity, as will collision response.

Pseudocode

```
p.velocity = (p.previousPosition - p.position) / timeStep
```

7.1.9 Collision detection & Response

Collisions and response with non axis-aligned planes and rectangles are implemented in this project, which are used to create a container for the fluid, and cuboid obstacles for the fluid to interact with.

Given a particle position, to calculate the distance to a plane the following is used [18]-

Pseudocode

```
dist = dot( (p.position - pointOnPlane), planeNormal )
```

If dist is less than the collision radius of the particle then the particle is colliding with the plane.

However, the above collision detection only considers an infinite plane, with no edges. For a rectangle, there must first be a check whether the position falls within the bounds of the rectangle, and then check the distance to the surface with the above.

For a rectangle where we know two vectors, from corner 1 (c1) to corner 2 (c2), and c1 to corner 3 (c3). We project the position of the particle onto these vectors, and if it falls within the length of these vectors, it is within the bounds of the rectangle.

Pseudocode

```
projection1 = dot( (p.position - c1), (c2 - c1) ) / (c2 - c1).lengthSquared
projection2 = dot( (p.position - c1), (c3 - c1) ) / (c3 - c1).lengthSquared
if projection1 >= 0.0 and projection1 <= 1.0 and projection2 >= 0 and projection2 <= 1 :
    checkDistanceToPlane()
```

Resolving Collision

The velocity is reflected in the direction of the plane normal and the particle position is set back to be outside the plane [19].

Pseudocode

```
p.velocity = p.velocity * -2 * dot(p.velocity, planeNormal) * planeNormal * bounceCoefficient
```

The bounceCoefficient is a constant between 0 and 1 that can dampen the velocity.

7.1.10 External Forces & Interactivity

External forces such as wind or gravity are applied to each particle's velocity -

Pseudocode

```
p.velocity += externalForce * timeStep
```

Interactivity

A small level of interactivity is added to the 2D implementation of the fluid where a left mouse click will attract fluid within a certain radius of the click, and allow it to be thrown. The right mouse click does the opposite, repelling particles that are within that same radius.

This is done by getting the position of the mouse while clicked, in world space. The linear kernel function used in the fluid calculations is also used here. Particles that are closest to the mouse click will be most strongly affected by the pull to its position. The pull or push is applied as follows -

Pseudocode

```
v = mousePosition - p.position p.velocity += v * ( 1 - (v.length / effectRadius) ) * strength
* timeStep
```

strength is a constant that will control how strong the pull/push is, however, if the left mouse button is clicked for a pull, the strength is positive, if the right mouse button is clicked it is negative. This is applied in the external forces step of the simulation.

7.2 GPU Implementation and Memory Synchonrization

As well as a fully CPU implementation, two GPU solutions were made for this project, both using Compute Shaders and SSBOs. In both of the implementations Compute Shader programs were created for each of the following steps -

- Advection
- Density, pressure and displacement calculations
- Viscosity calculations
- Collision detection and response

Separate SSBOs were created to store the particle data -

- Position
- Previous position
- Velocity
- Density
- Near density
- Pressure
- Near pressure
- Neighbours

All of the SSBOs matched their data types from the CPU side except for the Neighbours SSBO. In the CPU implementation, each particle stored a list of pointers to other particles. As the size of SSBOs must be allocated when it is

generated, a list data structure cannot be used, which is useful for a moving particles that will have a variable number of neighbours every frame. Therefore a maximum number of neighbours is defined, that should be high enough that there are never, or rarely, more neighbouring particles than can be stored, but low enough as to not waste memory.

The neighbours are stored as ints, which will be the index of a particle in the other SSBOs. The neighbour SSBO is not the same size as the other SSBOs due to storing MAX_NEIGHBOURS number of ints for each particle. The index of each particle's neighbours in the SSBO therefore starts at (particleIndex * MAX_NEIGHBOURS).

Before the neighbours array is filled, all elements are set to -1, which indicates there is no neighbour. The neighbours are then filled for each particle, starting at the beginning of the memory allocated for each particle. When particles are later running through their neighbours, when a value of -1 is found, we break from the loop as there are no neighbours remaining.

Constants such as kernel length etc. are defined as uniform variables in the shaders.

As each step of the fluid calculations must be completed before the next begins, this must be taken into consideration when invoking the shader programs. The function glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT), called on the CPU after each shader dispatch makes sure that previous OpenGL invocations are complete before the next begins, in this case specifically for the use of SSBOs.

7.2.1 Half GPU Implementation

The first of the GPU implementations keeps the neighbour searching on the CPU using the octree method with the remaining fluid calculations carried out on the GPU. This meant that very frame, the buffer containing the positions had to be written to memory on the CPU, so the positions could be used in the neighbour search. The neighbours SSBO would then be re-mapped with the appropriate data.

The neighbour search on the CPU was a limiting factor for performance. Reading and writing data back and forth from the GPU also incurred a heavy performance cost, making this an ineffective solution to speeding up the simulation.

7.2.2 Full GPU Implementation

The next implementation moved the neighbour search to the GPU, and changed the method from an Octree to the Simple Grid method from section 7.1.2. Two Compute Shader programs were added to fill the grid with particles, one to clear

the grid every frame, and one to run through every particle and add it to the appropriate cell in the grid.

An SSBO is added to represent the grid, which is structured much like the neighbours SSBO. The grid SSBO is of type int and is divided into cells. Like the neighbours SSBO a maximum number of particles each cell can store is defined. The first element of each cell stores the number of particles, and the next three store the x, y, z index position of the cell, with the next MAX_PARTICLES_PER_CELL storing the indices of particles in the cell. Therefore the size of each cell is (4 + MAX_PARTICLES_PER_CELL).

7.2.3 Memory Synchonisation

As many of the fluid calculations use neighbours attributes, there is potential for the same memory to be read from and written to by two threads at the same time. For example if two threads are looking at two particles which are neighbours to each other, or share the same neighbours.

Similarly when filling the grid with particle indices on the GPU. To fill the grid, each particle is looked at and the position in the grid is found, where it will then be added to the grid. Particles are added to the correct place in the grid SSBO depending on the number of particles currently in the cell, and so multiple threads may be trying to read this number of particles at once and then increment it by one when adding a particle.

Reading and writing to shared memory in parallel programming can lead to inconsistent results as what is written in one thread, may not be visible in another thread to read or write to [6]. This could lead to some strange behaviours in the half-GPU implementation where neighbouring particles are displaced and have velocity changed due to viscosity. For full-GPU implementation the problem was more grave as the grid could not be filled with the correct data, meaning no particles would have any neighbours stored, and therefore the fluid calculations had no effect.

Atomic Functions

One solution to this is to use atomic functions to alter SSBO data [5]. uints, ints, and floats (with the GL_NV_shader_atomic_float extension enabled) can be used in atomic functions. For example

atomicAdd(inout nint mem, nint data),

takes in a value mem and adds data to it. This can solve the problem when SSBO values need only be written to, however if they are read to, the original, unaltered value will still be read despite the atomic function.

Memory Barriers

GLSL contains memory barrier functions that are used to give control over the order of memory accesses [6]. The use of the function memoryBarrier( ) ensures that all memory transactions before the call must be completed before any other thread may make that memory transaction [6]. This can be used in combination with the function barrier( ), which will halt threads at that line until all threads have reached it.

Despite the available functions, this remained an unresolved problem come the end of the project.

8 Further Work

Resolving the memory synchronisation problems would be essential in continuing the work of this project. An exploration and analysis of other available GPGPU APIs, CUDA and OpenCL would also be an area that would merit further work. In terms of the fluid simulation, the elasticity area of Clavet et al. has yet to be implemented, as well as the stickiness of the fluid. This, as well as adding more comprehensive collision detection, for more primitive objects as well as complex polygonal shapes would greatly improve the effectiveness of the simulation in more practical applications.

References

[1] Bailey, M. OpenGL Compute Shaders,http://media.siggraph.org/education/conference/S2012_Materials/ComputeShader_1pp.pdf, 2012, [Online; accessed July 2016]

[2] OpenGL.org, Transform Feedback, https://www.opengl.org/wiki/Transform_Feedback, August 2016, [Online; accessed July 2016]

[3] Apple Inc., https://support.apple.com/en-gb/HT202823, August 2016, [Online; accessed August 2016]

[4] Khronos Group, glTransformFeedbackVaryings, https://www.opengl.org/sdk/docs/man/html/glTransformFeedbackVaryings.xhtml, [Online; accessed July 2016]

[5] OpenGL.org, Compute Shaders, https://www.opengl.org/wiki/Compute_Shader, May 2016, [Online; accessed June 2016]

[6] ARM Mali, Introduction to Compute Shaders, http://malideveloper.arm.com/resources/sample-code/introduction-compute-shaders-2/, [Online; accessed August 2016]

[7] OpenGL.org, Shader Storage Buffer Object, https://www.opengl.org/wiki/Shader_Storage_Buffer_Object, December 2015, [Online; accessed June 2016]

[8] OpenGL.org, Interface Block, hhttps://www.opengl.org/wiki/Interface_Block_(GLSL), June 2015, [Online; accessed July 2016]

[9] W. T. Reeves. Particle systems; a technique for modeling a class of fuzzy objects. ACM Trans. Graph., 2(2):91–108, April 1983.

[10] Monaghan, J. Smoothed particle hydrodynamics, Annual Review of Astronomy and Astrophysics, pp. 543–574, 1992.

[11] Muller, M., Charypar, D. and Gross, M. Particle-based fluid simulation for interactive applications, SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp. 154– 159, 2003.

[12] Clavet, S., Beaudoin, P. and Poulin, P. Particle-based viscoelastic fluid simulation, SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation, ACM, New York, NY, USA, pp. 219–228, 2005.

[13] N. Foster and D. Metaxas. Realistic Animation of Liquids. Graphical Models and Image Processing, 58(5):471– 483, 1996.

[14] Stam, J. Stable fluids., Conference Proceedings, Annual Conference Series, 1999.

[15] Jiang, C. Schroeder, C. Selle A. Teran, J. Stomakhin, A. The Affine Particle-in-cell Method, ACM Trans. Graph. 34, 4, Article 51, 10 pages , August 2015.

[16] Bridson, R., Fedkiw, R. and Muller-Fischer, M. Fluid simulation: Siggraph 2006 course notes, SIGGRAPH '06: ACM SIGGRAPH 2006 Courses, ACM, New York, NY, USA, pp. 1–87, 2006.

[17] Teschner, M. Heidelberger, B. Muller, M. Pomeranets, D. Gross, M. Optimized Spatial Hashing for Collision Detection of Deformable Objects, VMV, Munich, Germany, November 2003.

[18] Weisstein, Eric W. "Point-Plane Distance." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Point-PlaneDistance.html, [Online; accesed August 2016]

[19] Kelager, M. Lagrangian fluid dynamics using Smoothed Particle Hydrodynamics, January 2006.