2015

Modeling and rendering of fluid flows using the Lennard-Jones potential

Nicholas J. LeFave
Eastern Washington University

MODELING AND RENDERING OF FLUID FLOWS USING THE LENNARD-JONES POTENTIAL

A Thesis

Presented to

Eastern Washington University

Cheney, Washington

In Partial Fulfillment of the Requirements

For the Degree

Master of Science in Computer Science

By

Nicholas J. LeFave

Spring 2015

THESIS OF Nicholas LeFave

APPROVED BY

____ DATE ________

Dr. Paul Schimpf, Graduate Study Committee

____ DATE ________

Stuart Steiner, Graduate Study Committee

MASTER'S THESIS

Table of Contents

Table of Figures

MODELING AND RENDERING OF FLUID FLOWS USING THE LENNARD-JONES POTENTIAL

1 Introduction

The use of computational fluid dynamics holds many benefits for modeling real-world mechanical problems and for creating realistic effects for entertainment media. The following project examines the possibility of modeling realistic fluid flows using the Lennard-Jones potential and leveraging the power of modern graphics programming techniques by using the Graphics Processing Unit (GPU) for simulation of fluids. The project strives to solely use shader programs that run on consumer grade graphics processing hardware to model and render fluid dynamics governed by the Lennard-Jones Potential and to produce realistic looking results using screen-space rendering techniques. The use of screen-space rendering techniques requires multiple rendering passes to apply effects.

2 Background

2.1 Particle Systems

Particle systems are a computer graphics technique to simulate physical phenomena. Particle systems are only loosely defined in computer graphics and can be used to define modeling or rendering techniques, and certain kinds of animations. Many times the classification of a particle system depends on the application in which it is being used. There are, however, certain properties that are common to all particle systems. A significant property is that a particle system is a collection of one or more individual particles. Every particle in system has attributes that affect how or where a particle is rendered either directly or indirectly. Particles themselves are often graphical primitives like points or lines, although they are not limited to these primitive objects. Complex group dynamics, such as the flocking behavior of birds, have been successfully modeled using systems that manage multiple particles. The presence of some element of randomness in the actions of particles is another commonly seen property, and is used by the system developer to control the attributes of the particles. The introduction of randomness also allows varying behavior within a system. The attributes that are affected by randomness can include position, velocity, color, size, or the lifespan of the particles.

System particles in a scene are commonly generated by an emitter, an object whose sole purpose is to create and emit particles. Each individual particle is typically given an attribute called a lifespan. This lifespan determines how long the particle will stay in the simulation and is commonly used as a control measure to ensure that the application does not slow down because there are too many particles in a scene to calculate updates for. The particles that die are flagged to no longer be updated or to render. It is important to note that not every particle system has to have an emitter or that particles need a lifespan.

There are many potential applications of particle systems. They are frequently used to model chaotic phenomena due to having the ability to represent random behavior. One of the first examples of particle systems was the simulation of the behavior of fireworks. Particles in this type of simulation can model the trajectory of a firework object and, with additional particles, the subsequent colored explosions and smoke effects.

Modeling fluid flows using particle systems is not a new topic, however these particle systems have not until recent years been implemented on GPUs. Older particle systems ran entirely on a central processing unit (CPU) and were a pre-rendered sequence of frames played back at a faster rate to show a smooth fluid flow when modeling natural phenomena like water flows, smoke trails, or fire. Leveraging a GPU instead can lead to a nearly real-time simulation of the fluid. The drawback to particle systems is that they are computationally expensive due to the number of calculations controlling the interactions the particles have with each other as well as the environment in which they are rendered. Particle systems are classified as an N-body simulation. An N-body simulation is a term that defines a system in which there are N objects that interact with each other, while under the influence of other physical forces such as gravity. It is derived from the N-body problem that tries to predict the individual motions of a group of objects as they interact with each other under the effects of gravitation.

2.2 The Lennard-Jones Potential

Modeling fluid behavior of particles involves dealing with several types of forces affecting the particles. One type of force that particles experience during the modeling process is an interparticle force. Interparticle forces are the forces that particles exert on each other. A model of this behavior is the Lennard-Jones potential: a mathematical approximation of the interaction between a pair of atoms or molecules based on the distance separating those atoms or molecules. The Lennard-Jones potential describes the potential energy between two non-bonding atoms or molecules based on their distance of separation. The Lennard-Jones potential is not considered an accurate potential, however it is used extensively in computer simulations due to its computational simplicity. The Lennard-Jones potential is defined by the function

$$Lennard-Jones\ potential = \frac{k1}{r^{12}} - \frac{k2}{r^{6}}\ [2] \qquad (1)$$

The $k1$ and $k2$ variables in the function control the attraction and forces that particles exert on each other at a certain distance r from each other. The Lennard-Jones potential in this form is difficult to use in computer graphics simulations because the large exponential powers affecting the r variable are not convenient to calculate. To reduce the computational load, it is commonly reformulated to be in the form

$$Lennard-Jones\ potential = \frac{k1}{r^{4}} - \frac{k2}{r^{2}}\quad [2] \qquad (2)$$

As stated in Bridson [6], the exponential powers control how much the distance between particles affects the potential are popularly changed to being four and two instead of the original twelve and six. This change dramatically reduces the computational load. The Lennard-Jones potential can be applied to real world data but only realistically at the macroscopic level because of the computational overhead of modeling every molecule or atom in a fluid. This issue of scaling makes it inaccurate as a substitute for true fluid physics and requires the forces to be manually tuned for the results to look believable in a simulation.

The Lennard-Jones potential has a Big-O complexity of $O(N^2)$. For N number particles there are N^2 possible interactions that must be solved. The quadratic complexity of Lennard-Jones makes it a non-ideal candidate for real time interactive simulations without alteration. There are a few common alterations to Lennard-Jones to ensure the algorithm runs faster than $O(N^2)$. Asymptotically, the Lennard-Jones potential approaches 0 with large values of distance. One alteration is the introduction of a cutoff distance to take advantage of this property by saving unnecessary computation. Another common refinement to the Lennard-Jones potential is to use a grid system to find particles that would have influence on each other. Uniform spatial partitioning at a cutoff distance allows faster computation of the Lennard-Jones potential by eliminating most particles that are too far to influence each other. Instead of calculating the potential between all particles, it is only necessary to consider particles within the cell and the neighboring cells in the grid structure. By using a grid system, as long as the particles are evenly distributed and the number of particles per cell is held constant, interactions can be evaluated in linear time.

2.3 The Navier-Stokes equations

The governing equations for fluid flows are considered to be the Navier-Stokes equations, which are derived from the application of Newton’s Second Law of Motion to fluids. Navier-Stokes equations are formulated using an Eulerian frame of reference for fluid flows. The Eulerian frame of reference formulates equations by viewing fluid motion from the point of view of space through which parcels of fluid pass over a given period of time [2]. The other frame of reference for fluid flows is the Lagrangian frame of reference, which views fluid motion from the perspective of an individual parcel of a fluid traveling through both space and time [2].

The Navier-Stokes equations are a system of non-linear coupled partial differential equations. The Navier-stokes equations can also be reformatted into a single equation. The Navier-Stokes equations can also be simplified for different kinds of flows. One of these simplifications is for incompressible flows. Incompressible flows remove sound propagation or shock waves that may be present in a fluid. This simplification puts the Navier-Stokes equation into the form of

$$\rho\left(\frac{\partial v}{\partial t} + v \cdot \nabla v\right) = -\nabla p + \mu\nabla^2 v + \boldsymbol{f} \; [1] \qquad (3)$$

This mathematically represents a portion of Newton's 2nd law which states that a net force is equal to a mass multiplied by acceleration. The $\frac{\partial v}{\partial t}$ term is the unsteady acceleration of the fluid where $v \cdot \nabla v$ is the convective acceleration [1]. The sum of these accelerations is the total acceleration and ρ is the density of the fluid. Since density is mass per unit of volume, it is analogous to the mass in F= mass * acceleration. The right hand side of the Navier-Stokes equation is to show the sum of all forces that act on the fluid to produce acceleration. These forces include the pressure that the fluid is under, the viscosity of the fluid, and body forces. The $-\nabla p$ term is the pressure gradient and shows the non-linear effect pressure has on a fluid. Pressure is a surface stress that acts normal and inward to the surface of a fluid. It can be considered analogous to the normal force in solid mechanics, which is a force acting on an object in contact with another stable object. The $\mu\nabla^2$ term of the Navier-Stokes equations for incompressible flows depicts the viscosity of the fluid. The μ value is the dynamic viscosity of the fluid. It does not need to be constant. The ∇^2 term denotes the Laplace operator. Larger values of μ model more viscous fluids. Viscosity is a stress on a fluid that acts parallel to the fluid surface of a fluid. In solid dynamics this force is called the friction force. $\boldsymbol{f}$ represents body forces that are acting on the fluid. Body forces affect the entire fluid at once. A common body force is gravity. Gravitational force works exactly the same on a fluid as it does on solid objects in solid mechanics. Gravitation force on earth is calculated as $Fg = g * Mass\ of\ Object$ where g = 9.8 meters/sec^2.

The non-linearity of the Navier-Stokes equations makes these equations particularly difficult to solve, and in some cases a solution cannot be found. Being non-linear means that the equations cannot be expressed as a linear combination of first order differential equations which could be useful to put the equations into a form that is easily solved. The equations are non-linear due the presence of convective acceleration acting on the fluid. An example of this kind of acceleration would be a fluid passing through a narrow nozzle [9]. The compression of fluid within a nozzle causes the fluid to accelerate in narrower channels. This acceleration is based on the space they pass through not the force applied to project the fluid.

Due to the non-linear nature of these problems, they are commonly approximated through a numerical solution instead of explicitly solved as a closed-form solution. These kinds of problems do not have a standard method for solving them and require different techniques depending on the nature of the problem. The characteristics technique is one such approach of solving partial differential equations. The method of characteristics reduces a partial differential equation into a family of ordinary differential equations, which allows the solution to be obtained numerically via integration from some initial data.

One method of applying the Navier-stokes equations to a particle system is through the use of a method called Smoothed Particle Hydrodynamics or SPH. In order to solve the

Navier-Stokes equations it is necessary to create smooth, continuous fields from the properties of the particles at discrete locations in space. As stated in Bridson [3], this is essentially what SPH does. SPH will smooth discretely sampled attribute fields using smoothing kernels. In order to model a fluid flow as a particle system it is necessary to utilize a graphics library in order to create rendered images of the fluid.

2.4 The OpenGL Rendering Pipeline and OpenGL features

OpenGL is cross-language and multi-platform application programming interface (API) for rendering both 2D and 3D graphics. The primary use of the library is for interaction with a GPU for hardware-accelerated rendering. OpenGL can be viewed as a client-server model for processing commands on the GPU. The OpenGL API prepares and sends commands to the GPU to be processed in an application. The GPU then interprets and executes the commands. Modern OpenGL and graphics cards support the sending of programs to be executed on the graphics hardware. These programs are called shader programs; in OpenGL shader programs are written in the OpenGL Shader Language also known as GLSL. GLSL is a high-level programming language with syntax similar to the C programming language. The advent of shader programs has created the idea of the programmable GPU. Programmable graphics hardware has allowed more general-purpose computation on the GPU, as well as more elaborate graphics. This is possible because shader programs can be executed in various stages of the graphics rendering pipeline.

The first stage of this pipeline is called vertex processing. This stage handles all of the creation and movement of three dimensional points in a scene. In vertex processing, all of

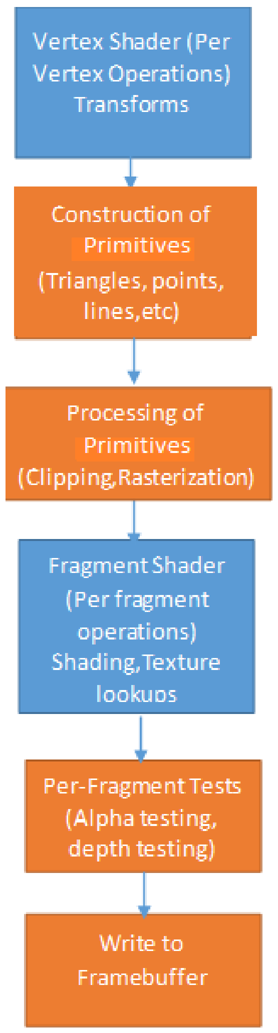
Vertex Shader (Per Vertex Operations) Transforms
Construction of Primitives (Triangles, points, lines,etc)
Processing of Primitives (Clipping,Rasterization)
Fragment Shader (Per fragment operations) Shading,Texture lookuos
Per-Fragment Tests (Alpha testing, depth testing)
Write to Framebuffer

Figure A Diagram of the OpenGL Rendering Pipeline

the vertices in a scene are processed as a list of vertices. A shader, called a vertex shader, can be bound and run in this stage of the pipeline. A vertex shader performs a processing task using the stream processor on the GPU. This means that the task is performed in parallel on the entire list of vertices. Vertex shaders are commonly used to apply transforms to the list of vertices e.g., to translate or rotate vertices around an axis or point. However, more complex processing can be done via shader programs in other stages of the rendering process.

The second stage of the pipeline is the construction of primitive objects. This is done by parsing a list of vertices into primitives. The commonly used primitive objects are points, lines, and triangles. In OpenGL, an application developer specifically states of the data is for points, lines, or triangles and the objects are constructed based on how many vertices is needed for each primitive. For example, setting the primitive type for the data for triangles assumes that every triplet of vertices that is read from the list is the position data for the 3 points representing a triangle. The primitive objects have been constructed after this stage finishes and proceeds to the primitive processing stage.

The primitive processing stage is done in static functions by OpenGL. After the consturction of primitive objects, OpenGL performs functions on the primitives. The most common functions performed in this stage are clipping and rasterization. The clipping function determines if a primitive object will be visible in the final rendered scene; if it is not, the primitive is not processed. Rasterization is a fixed function in OpenGL that converts the 3D vertices of the scene into 2D pixel (or fragment)

addresses of where those points will be on the computer’s display. The rasterization process leads directly into the next stage of the pipeline which is the fragment processing stage.

The fragment processing stage allows processing over all the fragments in an image after rasterization. The portion of a shader program that is written for this stage is called a fragment shader. When a fragment shader is executed, the shader program processes a list of all pixels in an image. The most common use of fragment processing is for shading surfaces of 3D objects to give the illusion of depth. Shading in this sense means selecting the color value for each pixel in the final image. Other common uses of fragment processing include applying filters to alter the quality of the final image. The use of fragment shaders also allow sampling another image for the composition of two images.

The rendering pipeline then performs several types of tests on fragments produced in the fragment processing stage. This is the per-sample processing stage of the OpenGL rendering pipeline. In this stage, the fragments created in the fragment shader are processed and their data is written to various buffers in OpenGL. In order to determine the specific fragments that will be processed in the creation of the final image, several per-fragment tests are performed. An example of one such test is the pixel ownership test which is run to make sure that fragments aimed at pixels not owned by OpenGL are discarded. This test is necessary because the default framebuffer is an external resource to OpenGL; it is possible that some pixels are not owned by OpenGL, but may be used by the Operating System or some other program. Another optional test that can be done in this stage is blending. Blending allows each of the colors in the fragment to be blended with the pixel color already present in the buffer to which the OpenGL program is writing. Combining this technique with RGBA colors allows alpha blending. The alpha value in an RGBA color controls the appearance of transparency on a particular color. An alpha value of zero makes the pixel completely transparent and a value of one causes complete opacity. Alpha blending is a technique that allows compositing of images using differing alpha values in the source and destination buffers. For example, an image of an object with a completely transparent background can be imposed on top of another image through sampling this image and discarding fragments that are completely transparent. After all of these stages have been traversed, the final image is constructed. The image is then stored in a framebuffer to be displayed on a screen or to be further manipulated.

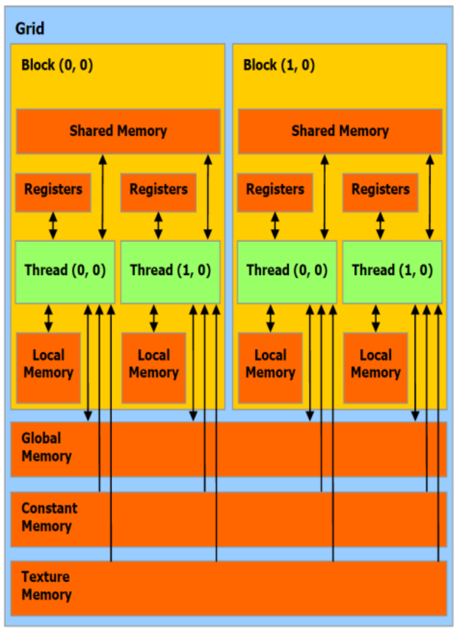

Figure B The GPU memory model as described by NVIDIA

2.4.1 The Memory Model of a GPU

OpenGL abstracts the allocation of memory on the GPU away from the developer. Even though the developer may not know where their data resides on the device, there are explicit places that data can be written via the client computer. As stated in The NVIDIA CUDA Programming

Guide [6], these memory spaces are global memory, constant memory, and texture memory as shown in Figure B. Global memory is addressable from any thread being executed on any GPU in the client system. Both the video device and the client computer can read and write to this portion of memory, but it is slower than any other type of memory on the device. Data written to the global memory space persists even after computation on the device. Constant memory space is similar to global memory space because it can be addressed anywhere. Constant memory space is very small in size, usually a few dozen kilobytes. The constant memory space is cached and can be accessed as fast as registers on the GPU. It also persists after execution of a program on the GPU. The last memory space is texture memory. Texture memory also resides in global memory space but is optimized for read operations giving slightly faster read performance than global memory space. Textures can be large in size and have random access. These memory spaces can be freed by the application that is using the GPU or in the case that the application using the GPU ends.

2.4.2 Data Management in the OpenGL Rendering Pipeline

Data in OpenGL is commonly stored in buffers known as buffer objects. Buffer objects are an OpenGL object that store an array of unformatted memory allocated on the GPU by the OpenGL context. One of the most common of these objects is called a vertex buffer object (VBO). These are buffer objects designed to store vertex data. Vertex buffer objects allow data to be transferred to the GPU for non-immediate-mode rendering. Non-immediate-mode rendering means that the data is sent to the GPU and is stored on the GPU for rendering [8]. The data does not need to be sent from system memory for multiple draws. The data in a VBO contains information relative to a set of vertices. For example, a VBO could contain the positions of a set of vertices, their color information, or the normal vectors for each vertex. The use of a VBO is a requirement for the initiation of a rendering pass using shader programs in OpenGL 3.3 or newer.

In order to accommodate more data being processed on the GPU, data can be sent as a texture. Using textures to send data allows the use of large blocks of memory to store a larger number of vertices and related information. A texture is an OpenGL object that contains one or more images in the same image format. The data in the texture does not necessarily represent an image to be displayed on the screen. Textures of data are sent to the video card using texture buffer objects. Texture buffer objects allow shader programs to access a large table of memory managed by a buffer object. Textures reside in a large global space of memory on the GPU.

There are other forms of storage besides buffer objects. One of the specialized forms of data storage is the frame buffer object (FBO). An FBO is a specific type of buffer object that allows a user to create and define their own framebuffer as opposed to the default framebuffer which is setup by OpenGL. Framebuffers usually represent a window or a display device as a collection of buffers which are write-only. These buffers are used in OpenGL for the rendering of an image to a display device or window. They control various properties of the image, including the colors of the pixels in the image. The use of a frame buffer object allows the definition of additional framebuffers to be managed by the OpenGL client code. The framebuffer also contains a special buffer called a renderbuffer. A renderbuffer is the specific buffer that holds an image. The difference between a renderbuffer-stored image and an image stored in a texture is that a renderbuffer-stored image is optimized for use as a rendering target, whereas texture may not be optimized. Renderbuffers are used primarily in cases where you do not need to sample from a produced image. Texture-stored images are optimized for image sampling. Frame buffer objects reference the images that are stored in renderbuffers or textures. As mentioned the use of textures for image storage is used for sampling an image. This allows sampling the image for additional effects to be applied to the image. Image processing in this fashion is commonly called a post-process effect, because it is additional processing pass after the initial processing of a single frame. It is also called multi-pass rendering because multiple passes of the rendering pipeline run to create a single final image for output to a display device. This rendering technique is used heavily for processing visual effects on images. Commonly, the effects rely on filtering to alter the perceived quality of an image by altering the color intensities in the original image.

Processing over a list of vertices in OpenGL does not keep any copies of the previous state of the vertices. Certain types of simulation will require the previous state of the vertices for the calculation of the forces involved. When it is necessary to have the previous state for all vertices using a feature in OpenGL known as transform feedback can produce this state data. All vertices processed in the vertex processing stage of the rendering pipeline are stored into a user defined buffer object. If many vertices are being processed, the best buffer object in which to store the vertex data is a vertex buffer object. In order to utilize transform feedback a few objects are needed in OpenGL. These objects are called transform feedback objects and transform feedback buffers. Transform feedback buffers are VBOs that are set up to store vertex data during vertex processing, while transform feedback objects are used as an abstraction to encapsulate information about the transform feedback. Transform feedback buffers and transform feedback objects stay in dedicated graphics memory. Transform feedback was necessary for the calculation of the Lennard-Jones potential interparticle forces. The transform feedback is used to store the previous frame positions for the vertices, which are needed to calculate the change in the Lennard-Jones potential for the next frame in animation.

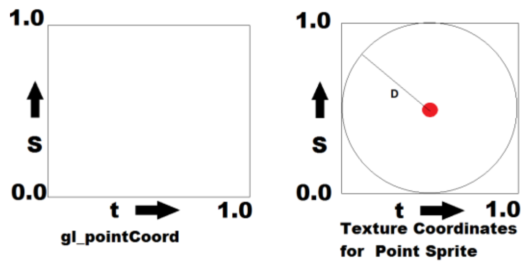

Figure C Diagram of gl_pointCoord mapping and the red dot of the second image shows the position in which we want to calculate distance from to construct the outline. Any pixels greater than a distance D should be discarded.

In order for particles in a simulation to look like pieces of water or anything other than a single pixel on the screen some sort of fragment processing is necessary to make them appear larger or change shape. One method of doing this in OpenGL is through the use of point sprites. Point sprites is a technique to swap the points after being rasterized to a quadrilateral which commonly referred to as a quad in computer graphics. This quad can have any texture or image mapped onto itself. In order to perform his mapping it is necessary to know where the fragment processor is within the rasterized point and map that to the same position in an image we want to map onto the quad. In OpenGL, it is possible to find where inside the rasterized point the fragment processor is by using a built-in variable called gl_PointCoord. It is possible to be within a point because a rasterized point in OpenGL may be comprised of several pixels or fragments, and therefore gl_PointCoord allows the traversal of these fragments. The gl_PointCoord variable gives two values, s and t, which give normalized coordinates of the current location within a point primitive. S gives a value of 0.0 to 1.0 that represents a location along the x axis that runs along the left and right side of the point which maps to the x axis within the context of an image. The value of t is a similarly normalized value but gives the location of where in the current point the fragment processor is at from the bottom to the top of the point which maps to the y axis in an image. To map an image onto the point sprite, it is necessary to calculate the texture coordinates for the image that is being imposed onto the sprite using the values s and t. The s and t values of gl_PointCoord directly translate to the texture coordinates in OpenGL, however it is necessary to find the distance of the current fragment from the center of the point sprite in order to draw the circular outline for each point sprite sphere.

3 Working in OpenGL

3.1 The Reformulation of the Lennard-Jones Potential

The work of this thesis strove to model fluids as particle systems. Previous research on using the Lennard-Jones potential for interparticle forces did show that the forces were particularly hard to adjust or fine tune in graphics simulations which is why the forces are commonly reformatted from their form shown in equation (1) into the form in equation (2). The Lennard-Jones potential modeling did lead to some particular issues of tuning that were not entirely discussed in previous works on the subject.

As mentioned previously, the values of r in the Lennard-Jones potential represent the distance between two particles. The exponential powers in the original formulation are computationally expensive. This causes the original formation to be extremely difficult to tune as the forces are affected greatly by distance. The forces were very sensitive to changes; the new formulation tries to compensate by lessening the sensitivity to changes.

During testing it was determined that even the common reformulation suggested by Bridson [3] was non-intuitive and more direct tuning of the individual properties of the interparticle forces was needed. The new formulation is

$$4 * \varepsilon * \left(\frac{\rho^4}{r^4} - \frac{\rho^2}{r^2}\right) \tag{4}$$

This form of Lennard-Jones is based off a popular formulation of Lennard-Jones which is

$$4 * \varepsilon \left[\left(\frac{\rho}{r}\right)^{12} - \left(\frac{\rho}{r}\right)^{6}\right] \text{ [7]} \tag{5}$$

The form in equation (4) is derived by lowering the powers of the potential and distributing the exponentional values of 4 and 2 instead of 12 and 6.
This reformulation allows individual tuning of the force between particles and simplifies the Lennard-Jones potential into accelerations instead of forces. The mass of the particles is the same for all particles in the simulation, so it is immaterial when reformatted in this way because all particles will have the same weight. ρ in this reformulation controls the distance of crossover between repulsion and attraction. ε is the maximum attractive acceleration that particles can experience and is a negative value.

The Lennard-Jones potential calculation is a large bottleneck in the performance of the overall program. The computational complexity of the Lennard-Jones potential is $O(n^2)$. Every particle in a simulation must calculate the force that every other particle exerts onto itself. To save computation many simulations will have a distance cutoff that will not calculate forces between particles far enough away from each other that the Lennard-

Jones potential is nearly negligible. This distance cutoff can cause discontinuity issues in the calculation of forces at the cutoff distance. The ideal solution is the implementation of a grid system to calculate the forces. A uniformly divided grid system subdivides the simulation space into cells and allows the execution of the potential to be fast by eliminating distant pairings of particles. However, it is possible to use any grid system to save computation for negligible particles due to large distances.

3.2 Implementation in OpenGL

3.2.1 Java and JOGL

This project has been developed in OpenGL using the Java OpenGL binding (JOGL). JOGL allows the Java programming language to access the OpenGL API. JOGL is responsible for communication with the OpenGL library, as well as setting up the rendering environment for OpenGL applications. The rendering environment includes the rendering window for displaying the frames rendered in OpenGL. The OpenGL client program in Java is also responsible for using JOGL to send data from the CPU to the GPU using OpenGL. JOGL also is responsible for managing the execution of all shader programs on the GPU. In order for a shader program to run, it must be checked for correctness, this happens by being linked and compiled by OpenGL. JOGL facilitates this process through the API. My project is a modification on the approach done by Simon Green at NVIDIA [3]. Simon Green's project modeled fluid flows using particle-based flows using the Direct3D and DirectCompute APIs. Although compute APIs like DirectCompute are a good use of the GPU, they lack compatibility across hardware. In order for my project to best use the GPU on many systems, all rendering and particle movement is done through a series of shader programs managed by the Java OpenGL client code.

3.2.2 Shader Programs

All drawing in the rendering window is done through shader programs of varying complexity. Shader programs at minimum have a vertex shader to utilize in the vertex processing stage of the rendering pipeline. A single shader program can also bind a fragment shader into the program as well. If both a vertex shader and fragment shader are bound to a shader program both shaders execute on a single rendering pass in OpenGL. Several of the shader programs used in this project are simple. These simple shaders are used to build a scene with objects or to display the interactions that the particles have on each other and the environment. This project also contains several shader programs that draw objects such as a background that provides contrast to allow the particles appear more easily in the scene. Other objects include a set of colored axis lines and a line rendering of the vessel, which collects the particles emitted by the particle system. The first shader program is responsible for the simulation of the physical forces bearing on the particles and the creation of the rasterized particles through point sprites. A diagram of the specific stages for this shader program is given in figure D.

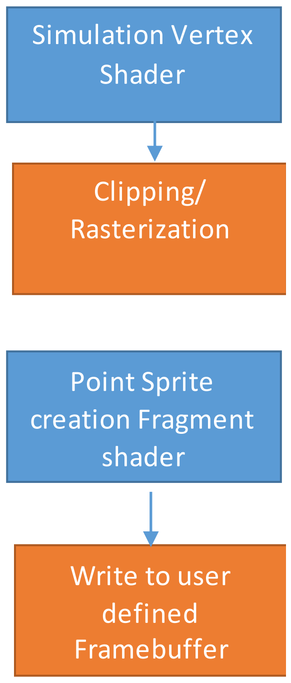
Simulation Vertex Shader
Clipping/ Rasterization
Point Sprite creation Fragment shader
Write to user defined Framebuffer

Figure D Outline of particle physics shader program

3.2.2.1 Physics Simulation of Particles

As mentioned previously, the first shader program is a shader that drives the physics simulation of the fluid and the rasterization of the particles. For the simulation of the particle physics, it is necessary to use the vertex processing stage of the graphics pipeline by using a vertex shader. This specific vertex shader is responsible for tracking the motion of each particle in the simulation. To accomplish this the center of each particle is modeled as a single vertex being processed in the vertex processing stage of the rendering pipeline. This shader updates velocity and position data for all particles through the time of the simulation. In order to update this data several pieces of uniform data are passed into the shader for the physics calculation. The uniform variables include the gravitational acceleration being applied to the particles and how many particles are in the total simulation. To properly calculate the Lennard-Jones potential it is necessary to have the previous position values of all particles. These values are stored in a Texture that can be accessed by subsequent shader calls. The function that allows lookup into a texture is called a TexelFetch. The TexelFetch function is a large source of slowdown in the overall program due to the fact that calculating the Lennard-Jones potential is an $O(n^2)$ operation and the operation of n^2 memory lookups is very costly. The simulation and the rendering of the particles were separated into separate shaders because the overall program performance suffered greatly from having a single shader and having separate shaders makes the logic of the programs easier. It was too much for a single shader to process the simulation updates and draw the vertices as particles. It is very common for the physics updates and rendering to be separated in particle simulations for tuning and debugging purposes. The shader programs use the vertex processing stage to manipulate the position and velocity of each particle in the simulation. As mentioned previously these processed vertices are then passed into a fixed function stage in which the vertices are clipped and rasterized.

3.2.2.2 Point Sprite Rendering of Particles

As shown in figure D, the next portion of the shader program handles the process of the actual rendering or drawing of the particles. These particles are rendered as point sprites using a fragment shader. In order for particles to be rendered looking like masses of water or fluid it is necessary to render vertices as point sprites. As mentioned point sprites are a technique that can be used in OpenGL to replace a set of vertices that would be rendered as points with rectangular objects called quads. These sprites allow the display of a texture in place of a rasterized point. For the purpose of this project this shader transforms the rectangular point sprites into what looks like a sphere centered on the position of each vertex.

3.2.2.3 Shaping the Particles

To give the appearance that each vertex is a particle of water, it is necessary to make a circular outline for the texture used by the point sprite. This can be done procedurally with a fragment shader to make the texture for the point sprites. With proper coloring of these textures they will appear as spheres with transparency. In order to create the circular outline for these spheres some fragments or pixels are explicitly discarded and not processed by the fragment processor. This requires the position of the particle to help find the position of a pixel relative to the center of the point sprite.

This is done by using the fixed gl_PointCoord function which gives the coordinates of where the fragment processor currently is in the current point sprite. These coordinates are translated to make the origin of coordinate system center around the vertex position. In order make the circular shape of the sphere we want to discard any pixels found at a particular radial distance from the center of the sphere. To calculate the distance from the center of the point sprite we first the location of the current fragment using gl_PointCoord and translate these coordinates to be centered on the vertex.

Creating a two dimensional vector of these positions and performing the dot product of the vector with itself will give the scalar distance of the current pixel is from the center of the Sprite. A simple test checking if the radial distance is greater than one is all that is needed to determine which pixels should be discarded, since the coordinates are normalized. After the creation of the Point Sprite outlines, an image is created with each vertex changed into the circular outlines of the particles. The color data of these point sprite spheres is actually encoded as the position data for each particle. The selection of the color of the point sprites will be done in an additional rendering pass. In order to perform extra rendering passes and image effects, the image is saved into a texture via a user defined framebuffer.

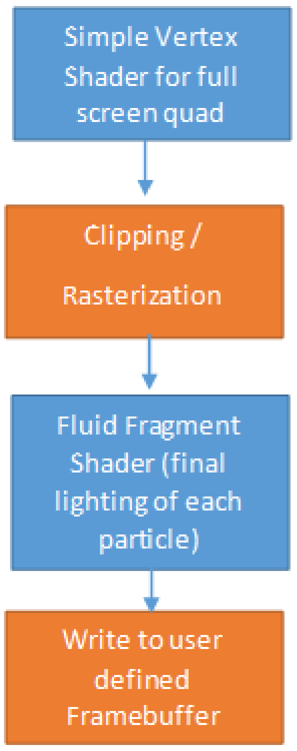
Simple Vertex Shader for full screen quad
Clipping / Rasterization
Fluid Fragment Shader (final lighting of each particle)
Write to user defined Framebuffer

Figure E Outline of stages used by the fluid rendering shader

3.2.3 Multi-Pass Rendering

After the execution of the previous shader programs, the particles are rendered as circular sprites instead of single pixel points on the screen. The color used to fill these point sprites is not sufficient to produce a realistic rendering of the fluid and is actually position data for each particle. In order to make the fluid look realistic, multiple rendering passes are needed to color and apply effects to the point sprite spheres.

3.2.3.1 Shading of Fluid Particles using Phong Shading

The first of these passes runs a shader program to color the individual particles so they appear more realistically shaded. This shader calculates the color of the fluid by using Phong shading. The Phong lighting model (also known as Phong shading) simulates light as it reflects off a surface. Phong shading takes into account several features of lights in a scene. The first property modeled is the ambient light in a scene. Ambient lighting is a fixed-intensity and color lighting in a scene and can be thought of as a background light that appears to come from all directions. The intention of ambient light is to model light that strikes an object after being reflected off other objects in a scene. Ambient light is modeled as

$$Ambient\ light\ = Ia * Ka\ [3] \tag{6}$$

where Ia represents the intensity of one wavelength of ambient light that strikes the object and Ka represents an object's reflection coefficient for the same wavelength of light. Separate equations are necessary to maintain for red, green, and blue wavelengths [3].

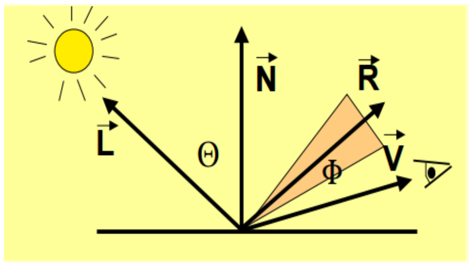

Figure F Diagram of vector quantities needed for the calculation of Phong Shading [6]

Next is the diffuse reflection, modeled through Lambertian reflection. Diffuse reflections are the reflections seen on most real world materials. Most surfaces are made of materials that are not completely reflective. The materials will absorb some of the light that their surfaces receive and this must be considered in the final calculation of surface shading. Lambertian reflection also allows 2D projections of an object to appear as a 3D object by creating shadows from the light sources in a scene. Lambertian reflection applies Lambert's Law which states that the light intensity on a surface depends on the angle of the incoming light. [3] Figure F depicts the vector quantities needed for the calculation of the diffuse reflection. The Vector $\vec{\mathrm{L}}$ is a unit vector showing the direction of the light source from the surface. Vector $\vec{\mathrm{N}}$ is the normal vector from the surface that is to be shaded. Vector $\vec{\mathrm{R}}$ is a unit vector that signifies the direction of the light's reflection off the surface. The final vector, known as the view vector, which is needed for the calculation of diffuse color, is Vector $\vec{\mathrm{V}}$ which represents the direction of the viewer in relation to the surface. The calculation of the intensity of diffuse color for N lights in a scene uses the aforementioned vector quantities and is given by the equation

$$Id = \sum_{i=1}^{n} \frac{Kd\,(\vec{\mathrm{L}} \cdot \vec{\mathrm{N}})}{d} \text{ [3]} \tag{7}$$

The value Kd is similar to the Ka constant in the calculation of the ambient light in a room. Kd represents the reflection coefficient for the diffuse color. $\vec{\mathrm{L}} \cdot \vec{\mathrm{N}}$ varies the intensity of the diffuse color depending on the angle of the light source to the lit surface. The equation divides these values by constants d and k for attenuation of the light based off distance. The d value represents the distance from the light source to the eye or camera in the scene. [6]

The final portion of the Phong lighting model models the specular reflection on the surface. This creates specular highlights on a surface from the lights in a scene. In order to calculate the influence of specular lighting it is necessary to use the equation

$$Ispec = \sum_{i=1}^{n} IiKs(\vec{\mathrm{R}} \cdot \vec{\mathrm{V}})^{n}/(d) \text{ [3]} \tag{8}$$

The equation states that specular color intensity is equal to the sum of the intensity Ii of each light in the scene and its Ks constant which represents the reflection coefficient for the specular color of the light. This value is dependent on the viewing angle, which can be represented as the dot product of the $\vec{\mathrm{R}}$ and $\vec{\mathrm{V}}$ unit vectors. As mentioned before, $\vec{\mathrm{V}}$ represents the vector from which the viewer sees the object and $\vec{\mathrm{R}}$ is a unit vector that shows the direction of the reflection off of the surface. The value of n is used as a shininess factor of the surface. This factor is used to control the specular reflection's glossiness level. The final lighting equation sums the intensity from ambient, diffuse, and specular lighting for all N lights in a scene into a single equation. Thus the final equation is

$$Itotal = IaKa + \sum_{i=1}^{n} \frac{IiKd\,(\vec{\mathrm{L}} \cdot \vec{\mathrm{N}}) + Ks\,(\vec{\mathrm{R}} \cdot \vec{\mathrm{V}})}{d} \text{ [3]} \tag{9}$$

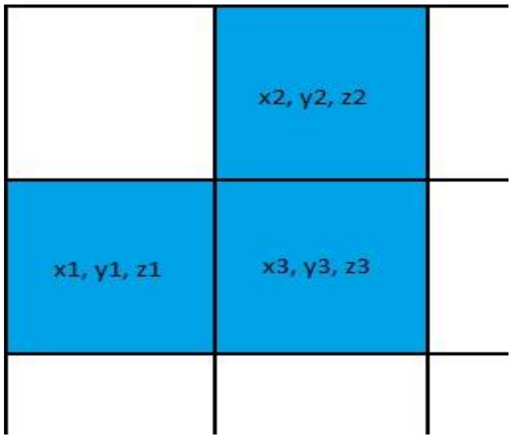

Figure G Shows pixel values in the image. Each pixel contains the position data for the particle. These using 3 of these values allows the calculation of a normal for a single pixel.

The final lighting equation is applied via a fragment shader to the particles. To do this we take the view space positions of all of our particles that have been saved into the texture and use them to apply specular shading highlights based on the light sources in the scene. To correctly shade a specular highlight, it is necessary to use a normal vector from the surface that would have the appearance of the highlight. This normal vector represents the direction that light reflects off the surface and to the eye of the viewer.

3.2.3.2 Calculating Normal vectors for a Surface in a Rasterized Image

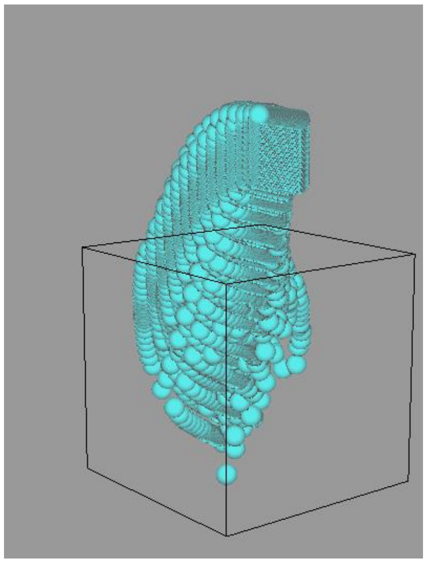
Figure H Sample image of the lighting calcuation per particle

To create the normal vector for doing the Phong shading calculation for each pixel in the image it is necessary to find the cross product of two vectors. These two vectors are constructed with the view space positions stored in an image. This is done by using a pixel in the image with two of the current pixel's neighboring pixels. The x,y, and z position values are stored in the red, green, and blue color channels respectively. In order to create the normal, it is required to create two vectors sharing a common vertex and then perform the cross product of these vectors. This will create a normal vector for that particular pixel in the image. Using the lighting position data and the normal vector created for every pixel in the image, it is possible to apply Phong shading to every particle in the scene allowing definition of final color information for the fluid. Figure E outlines the different stages of the rendering pipeline that are utilized in the creation of the image of the particles having lighting and coloring calculations. Figure H is a sample image of the output of the particles after Phong shading has been applied to the point sprite particles.

3.2.3.3 Rendering to a Full Screen Quad

Although the lighting calculations have been placed and are now ready to be written into a framebuffer, the process of writing the image has changed. The vertex processing stage of the rendering pipeline is only necessary as it is required to create a valid shader program. It is not necessary to use the Vertex Processing Stage to perform transforms to a set a vertices. Instead of processing vertices, we have a rasterized image that needs to be displayed to the screen. This is done by a technique known as rendering to a full screen quad. Rendering to a full screen quad creates an image onto the display device from a sampled image. As in the previous shader, both the vertex and fragment processing stages are used. The vertex processing step is essentially a pass-through stage because it is no longer needed to process over a list of vertices that represent the positions of each particle in the scene. Instead the vertex processing stage processes a different list of vertices that define the boundary viewable area of the virtual camera. This is viewable area, called a full screen quad, and it is a specific quadrilateral that is positioned to be in the entire viewable area of the virtual camera. It allows scenes that have been rendered to a texture to be displayed on screen via a fragment shader that samples the texture and outputs the pixels to the display device. The rendering process was not entirely finished after the calculation of the color data per pixel for the image with the point sprite spheres as they do not look realistic. Despite having the coloring of water, the resulting frames do not resemble a smooth fluid surface due to distinctly defined spherical edges associated with each particle. An additional rendering pass is necessary to blur the particles together into what looks like a fluid surface.

3.2.3.4 Image Filtering

As detailed in the Simon Green presentation at GDC 2010 [5], to make a fluid created from point sprite particles appear more realistic, it is necessary to apply a blurring filter to the image of the point sprites. Image blur gives the particles the appearance of a fluid. Image filters, such as the Gaussian filter used in this project, commonly sharpen or blur images for the removal of artifacts in the source image. The Gaussian filter is built from the Gaussian function, which is of the form

$$f(x) = a \exp\left(-\frac{(x-b)^2}{2c^2}\right) + d \tag{10}$$

The variables a, b, c, and d are arbitrary constants. The Gaussian Function expresses a normalization distribution and is the characteristic bell curve-shaped function. An example graph of a three-dimensional Gaussian function is given in figure I. An example of a two-dimensional Gaussian blurring filter is the equation

$$G(x,y) = \frac{1}{2\sigma^2} e^{\frac{-x^2+y^2}{2\sigma^2}} \tag{11}$$

In this formula, the terms x and y control the distance from the origin in the horizontal direction and the vertical direction respectively. σ is the standard deviation for the Gaussian distribution.

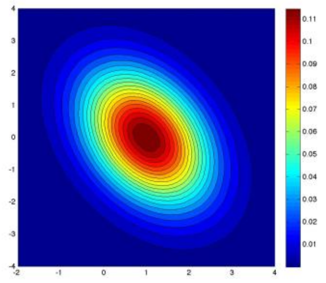

Figure I An Example graph of a two –dimensional Gaussian function

The Gaussian blur filter is commonly used to reduce image noise and reduce detail. The strength of a Gaussian blur filter is a term that defines the strength of the normalization of the averaging function. A higher value for strength will produce a blurrier image through the manipulation of the Gaussian function. A single pixel in the image will have the color value be set to the weighted average of neighboring pixels. This area is called the kernel filter and can be adjusted in size. The Gaussian filter is considered a separable filter because it can be applied to a two dimensional image as two independent one dimensional calculations. Separating the filter is computationally cheaper, but in the current state the project does not have two separate filters: this would require another shader program to be run over the image. In order for the current solution to be optimal the filtering should be separated. The Gaussian function uses a kernel function for the distribution of the weights it assigns to pixels. For the sake of speed these weights are pre-computed within the filtering kernel. However silhouette edges can appear in the final rendering, because the filtering kernel could be averaging pixels that do not have color values of the particles. Silhouette edges are a collection of points whose outward surface normals are perpendicular to the view vector. Blurring across silhouette edges causes artifacts due to the discontinuities along such edges. Solutions to this issue can include different types of blurring filters or preserving edges in another texture and combining the results.

One potential solution to these silhouette edge artifacts is to use Bilateral filtering instead of Gaussian blurring [5]. A Bilateral filter is better at preserving edges by detecting sharp changes of color in an image and changing the weights and filter kernel to ensure there is no averaging across these boundaries. This is a non-separable filter and is slower than a separable Gaussian filter.

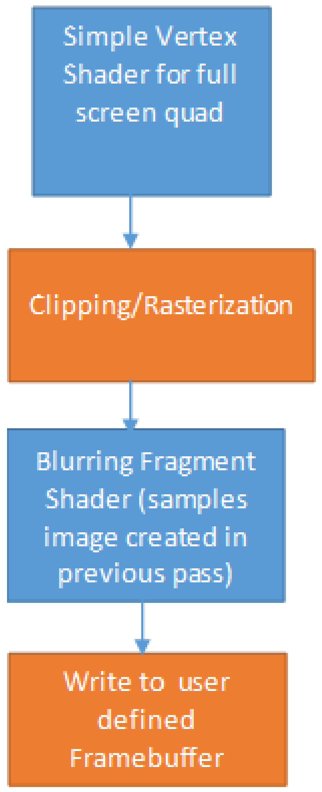
Simple Vertex Shader for full screen quad
Clipping/Rasterization
Blurring Fragment Shader (samples image created in previous pass)
Write to user defined Framebuffer

Figure J Outline of the Rendering Pipeline stages for a Gaussian filtering shader program

3.2.3.4.1 Gaussian Blur Filtering using GLSL As stated previously, in order to construct a surface for the fluid using particles, it is necessary to apply a blur filter to the image of the point sprite particles. In this project a Gaussian blur filter is applied to this image using another rendering pass and using the fragment processor to apply the filter to the image. The outline of the various stages of this shader program is shown in Figure J. As with the final lightning calculation shader, the vertex processing stage is essentially a pass-through the vertex processor using vertex positions for the full screen quad onto which the blurred image will project. The fragment processing stage applies a Gaussian blur to the image of the point sprite particles, and alpha testing is then used to separate and discard the rendered background from the particles to ensure that the background color does not become

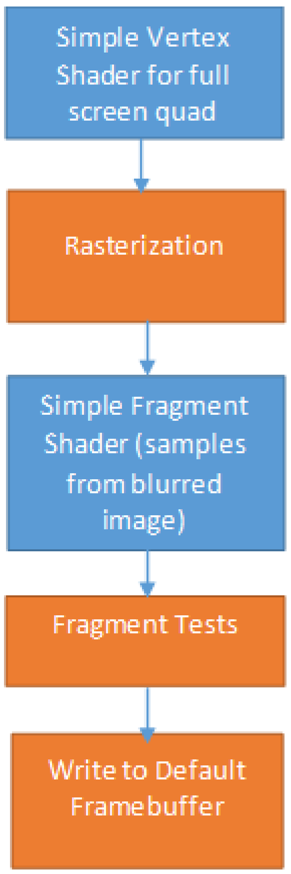
Simple Vertex Shader for full screen quad
Rasterization
Simple Fragment Shader (samples from blurred image)
Fragment Tests
Write to Default Framebuffer

Figure K Stages needed by the final shader program to write the blurred image into the framebuffer.

blurred into the particles. The final settings for the Gaussian blur include a filtering kernel that is a nine-by-nine kernel with 4 as the strength of the kernel. These settings for the blur gave the best mix of speed and quality for rendering the fluid surface. For testing purposes, both bilateral and Gaussian blurs were used. Gaussian blurring did not produce artifacts if the blurring happened after the final lighting and shading calculations for coloring the particles were completed. Switching the stages, or doing the blurring before the final shading calculations, did create artifacts on silhouette edges with a standard Gaussian blur. This is where my project deviated from the method described by Simon Green in his presentation [5]. Green applied his filtering before calculating his final lighting and shading values for the fluid using a separable bilateral filter. Attempts to use a bilateral filter for blurring did not solve the artifacts created from silhouette edges when blurring occurred before final lighting calculations. However, bilateral filter also did not produce artifacts when the blurring process happened after the final shading of the fluid surface. As noted in Figure J, the shader program for blurring of the image uses both a vertex and fragment shader. The vertex shader is like the previous the vertex shader used in the shader program for the final lighting calculations; it functions only for the use of a full screen quad for rendering. The fragment shader samples the image with the final lighting calculations and applies a Gaussian filter to blur the appearance of the particles. The image is once again saved to a framebuffer so the last rendering pass can make the final display image.

The final image is read to the screen after being composited with the default framebuffer in OpenGL. The outline of the stages used in the rendering pipeline is given by Figure K. The default framebuffer holds an image of the background with our box vessel used to collect the fluid. Similarly, to the rendering technique for both the final lighting and blurring shaders a full screen quad is used to render the image of the smoothed particles. Using alpha blending, the image of the particles is imposed on top of the background image, thus creating the final image that will be displayed on the screen. Alpha blending is a technique that uses a value of transparency to combine two separate images into a single image by mixing pixel color values between the two images using the alpha value for each pixel. The weighting of the color is determined by the alpha values in both images. An example image of the final output of the fluid is given in Figure L.

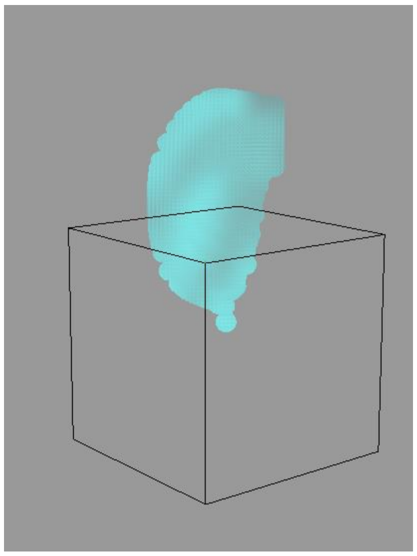

Figure L Final results of rendering process after blurring

4 Conclusion

In conclusion, it is possible to generate fluid flows using only OpenGL and the OpenGL shading language. Despite being able to render what appears to look like flows solely using the Lennard-Jones potential, it is not an adequate model for giving realistic results on its own. More complicated physics models like the Navier-Stokes equations appear to be necessary to give fluid animations realistic appearing flows. In order to implement Navier-Stokes based simulation other technologies are needed other than using exclusively shader programs. Another shortcoming when using OpenGL and GLSL is that they are unintuitive in this case due to the nature of the types of calculations being done in the shader programs. This is because shader programs are designed for the graphics pipeline and fit nicely for rendering processes. They are not intuitive for general purpose GPU (GPGPU) calculations and modeling physics. Other technologies like NVIDIA's CUDA or the OpenCL programming framework are common for these kinds of simulations, due to lacking the necessity of a deep understanding of the graphics pipeline or a graphics library. Also, these frameworks are similar syntactically to writing in the C programming language which many developers are familiar with. Many of these frameworks also have spatial partitioning structures that can be used for the speed up of the calculations of the Lennard-Jones potential. Due to these features these technologies are more useful for generalized programming using the GPU than GLSL. GLSL has recently added support for compute shaders that could potentially be able to do the physics calculations required for SPH. GLSL would be a good supplement to OpenCL or CUDA; since it is tightly coupled to the graphics rendering pipeline it can handle the rendering and shading processes. Using the GPU for heavy calculations and rendering is a major problem for OpenGL due to how rapidly graphics hardware has changed. In light of this the Khronos Group (developers of OpenGL) and have begun work and announced a successor API to OpenGL at GDC 2015 called Project Vulkan. Project Vulkan is a substantially different API than OpenGL that gives more low-level access to the hardware in a C-like language for both rendering and generic calculations on the GPU. The low-level access to the hardware shows significant improvement to real time graphics applications. Project Vulkan has already gained support from hardware manufactures and game development studios.

Although there is a visible fluid flow using GLSL shaders, the overall performance is fairly low. A simulation of 60,000 particles interacting in the vessel animates at 3 to 6 frames per second (FPS) depending on how many of the particles have been emitted. This is not ideal as animations for many real-time applications would like to animate at a minimum of 15 frames per second. Animations lower than 15 FPS appear very slow and choppy to the human eye. Many modern video games in contrast, try to run at 60 FPS, as this is the upper limit what most human eyes can perceive. At that frame rate, all animations seem lifelike and smooth. Most video game engines would not try to use 60,000 particles at a time due to the computational complexity.

5 Future Work

There are many aspects of this project that are considered for future work. For one, this project fell short of utilizing full SPH simulations. SPH simulations of fluid flows can create much more realistic looking results. Therefore, as future work creation of SPH simulations using the OpenCL framework for the fluid solver would be more beneficial than extension of existing shader programs. The current rendering code would be sufficient for the updated solving method because OpenCL can be used in conjunction with OpenGL for the rendering process. The code presented in this thesis is also not optimized. The use of SPH would entail the use of a grid system for solving the simulation which would increase the compute performance of the simulation. The optimization shader program code as well as communication between the GPU and CPU a difficult problem but could be beneficial to examine to increase performance. Due to familiarity with JOGL and Java programming, the JOGL library was initially chosen for this project, however due to issues with cross platform compatibility in JOGL across hardware vendors of GPUs, conversion of the codebase to C++ and standard OpenGL could be useful for better multiplatform support of this program. In the future this project will be translated into the Vulkan API as it already is seeing massive support from graphics card vendors and game developers, however the Vulkan API is currently not near a full public release.

References

[1] Bakker, Andre. "Computational Fluid Dynamics." Lecture 4 - Classification of Flows. N.p., 2006. Web. 18 March 2015. Retrieved from http://www.bakker.org/dartmouth06/engs150/04-clsfn.pdf

[2] Bridson, R. 2007. FLUID SIMULATION SIGGRAPH 2007 Course Notes. Extrapolation (2007), 1–81.

[3] Clark, Bill. 'Shading and Lighting'. 2012. Presentation.

[4] Cohen, J. M., Tariq, S., & Green, S. (2010). Interactive fluid-particle simulation using translating eulerian grids. Paper presented at the *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Washington, D.C. pp. 15-22. doi:10.1145/1730804.1730807

[5] Green, S. (2010). Screen Space Fluid Rendering For Games. Presented at the *Game Developers Conference 2010*, San Francisco, California. 1-39 Retrieved January 10, 2014 from NVIDIA Developers: http://developer.download.nivdia.com/presentations/2010/gdc/Direct3D_Effects.pdf

[6] NVIDIA CORPORATION. 2015. Programming Guide: CUDA Toolkit Documentation. CUDA C Programming Guide. Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[7] Poling, B, Prausnitz, O'Connell. The Properties of Gases and Liquids 5th Edition

McGraw-Hill, New York, 2001, 11.6

[8] Shriener, D. et al. 2013. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3. Addison-Wesley Professional.

[9] Stern, Fred. “Fluid Kinematics” Chapter 4 N.p., 2013. Web 18 March 2015. Retrieved from http://user.engineering.uiowa.edu/~fluids/posting/lecture_notes/chapter4.pdf

Bibliography

Chentanez, N., & Muller, M. (2010). Real-time simulation of large bodies of water with small scale details. Paper presented at the *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation,* Madrid, Spain. pp. 197-206. Retrieved from http://dl.acm.org.ezproxy.library.ewu.edu/citation.cfm?id=1921427.1921457

Drone, S. (2007). Real-time particle systems on the GPU in dynamic environments. Paper presented at the *ACM SIGGRAPH 2007 Courses,* San Diego, California. pp. 80-96. doi:10.1145/1281500.1281670

Kruger, J., Kipfer, P., Konclratieva, P., & Westermann, R. (2005). A particle system for interactive visualization of 3D flows. *Visualization and Computer Graphics, IEEE Transactions on, 11*(6), 744-756.

VITA

Author: Nicholas J. LeFave

Place of Birth: Spokane, Washington

Undergraduate Schools attended: Eastern Washington University

Degrees Awarded: Bachelors of Science, 2012, Eastern Washington University

Honors and Awards: Graduate Assistantship, Computer Science Department, 2012-2014, Eastern Washington University

Professional Experience: Internship, DigiDeal Inc, Spokane, Washington, 2012