Fluid Simulation for Fire Particle Systems using Octet

Louis Bennette

University of Goldsmiths
Msc Computer Games and Entertainment - 2016 - 2017
18th September 2017

Table of Contents:

Section 1: Introduction

As games become ever more immersive and realistic visually, with high definition imagery and smooth flowing action, and in particular for virtual reality applications, the degree of 'reality' depends on the technical ability of the software to show objects and scenes behaving in an apparently natural way.
Previous technology that relied on images or "sprites" and simple newtonian simulations of motion and form is no longer good enough to convey a realistic scene to match the now standardised expectation of high definition imagery.
One example of this is the use of flames in a scene. For flames to appear realistic they should interact with the scene, for example in the consumption of a 'fuel' and interaction with flammable and non-flammable objects.

It's important to stress that the flame does not need to actually behave as a real flame would under the exact same conditions in real life. It only needs to create the illusion that the flame looks real.
A game's theme and graphical style helps to sell the individual components as a part of their own plausible world, achieving a good level of convincing plausibility increase the immersive nature of a game. This means that immersion is not always tied to realism. However, adding elements of realism enriches the player's surroundings. This is also true for the narrative content and audio scapes of games.

The goal in the project was to explore programming techniques that can be utilised to produce realistic flames. The flames need to be computed and rendered in real time, modern games typically have a 60 frames per second refresh rate. This mean that whatever techniques are used need to be fast and efficient to allow for gameplay and mechanics. Additionally, the flames must look realistic and help to provide an immersive experience to the player. To achieve this the artificial flames would need to have characteristics of a real flame, for example a blazing rapidly coiling flame we would expect it to feel hot and dangerous whereas a candle flame should appear laminar, flowing and contained within a reasonable space. Making the player feel as though the flames have these properties is complementary to a well designed game and provides an entertaining experience.

I decided to tackle the problem of making a flame look convincing with the relatively limited resources of a regular PC / gaming machine. Flame generation for pre-rendered movie CGI applications can use any amount of CPU horsepower the designer chooses, with the only cost being the time taken to make each scene, and the possibility to throw more computers at the problem if the time is too slow. In a real-time game there is only

the time of one playback frame, in the order of tenths or hundredths of a second and the amount of computation feasible is quite limited.
An aspect of this paper is the comparison of CPU and GPU 'costs' i.e. the amount of hardware resources needed and available vs the quality of appearance of the rendered scene. It is possible that present hardware ability is not yet able to perform a fully real-time synthesis of a flame that looks and behaves well and leaves the computer with sufficient resources to run other functions for most games. In most cases flames are not involved in the mechanics or narrative of games, they are visual component that do not warrant a realistic simulation.

The experimental part of the research has been conducted using the open source C++ game engine "Octet" written by Andy Thomason (Thomason A. 2016). This environment provides direct access to openGL as well as convenient wrapper classes for building a game. There is also openCL support which has been fundamental to part of the experimentation and optimisations required by the project.
It will be shown that it remains necessary to employ shortcuts and tricks to build a scene and that pure algorithmic synthesis, while capable of remarkable realism, cannot deliver on most current hardware in continuous real-time.

Realistic flames are important to simulate if the output they produce is entertaining for the player. A realistic wind simulation is justifiable and desirable for a game studio if gameplay and narrative can be derived from the simulation, for example a kite game could benefit from a wind simulation. With this in mind, if the context of the advanced simulator has entertainment value worth the computational costs, it is a valuable feature to implement.

Section 2: Hypothesis

The research conducted is based on how to develop a realistic realtime fire. Principally, this work looks into the movement and colouration of flames, excluding the emissive properties of flames as a light source. Additionally, the involvement of fuel was explored, but only as a measure for where the reaction happens in space; the source of energy being important in the movement of the flame.

Work on this project originally started with the intention to make the fire feel as though it was hot. This means it would need to feel voluminous, to move stochastically and also to have the right colouration and convincing scale.

Section 2.1: Particle system

Particle systems in games are large sets of particles, in a scene animated with some transformations over time. A particle is typically a point or a quadrangle with a texture image displayed. These quadrangles tend to have their plane oriented at 90 degrees to the scene camera, and are said to be always facing the camera. These are termed billboard particles, other types of particles also exist, such as trail particles, point sprites or even three dimensional models.

For this research project, billboard particles are used. This is because they are computationally cheap. Typically particle systems use simple animation to move the particles around. For example a typical fire and smoke effect is done using newtonian acceleration. The current velocity of each particle is tracked and is modified each game loop using a newtonian equation like: $\boldsymbol{V = U + at}$
Where V is the translation performed on the particle and the particle object is updated with the new current velocity.
Octet provides these particle systems although they are not fully optimised for the task of rendering realistic flames. This is developed further in Section 3.1.

Section 2.2: Light in real Flames

Light from a real flame is a natural effect produced by an excited electron releasing it's energy and returning to a lower stable energy state. The energy released is in the form of a photon with a wavelength determined by the change of the electron's state.
Specifically in chemistry an electron can move around inside physical regions around an atom's' nucleus known as electron orbitals. Depending on the element or molecule different number of electrons will exist in different orbital regions around the nucleus. This more thoroughly covered by. (Khan Academy n.d.)

Below we can see an illustration which explains the phenomenon of light being released by an electron. When energy is introduced an electron can be promoted to a higher electron orbital. The electron can't remain in a higher orbital and as it falls back down to a lower orbital it releases its stored potential energy in the form of a photon. The diagram below illustrates how different orbitals correspond to different electron states and the leap between orbitals corresponds to the wavelength of the released photon. Different elements have different electron orbital configurations these influence the output colour, or wavelength, in different amounts.

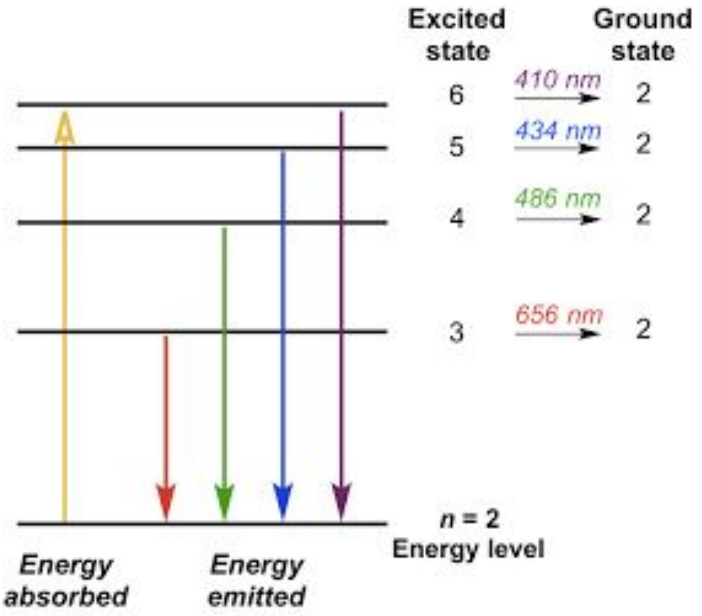

***Figure 1**: Diagram of photon wavelength emission for different electron orbital leaps*

Colour in fire is created by the amount of energy input in the system as well as the type of element or molecule that is being heated. With this knowledge, the colour of the flame could be approximated by tracking the energy of the system. A fuel model could be used to track the fuel type and the fuel could provide colour variations in a region. This would involve a colouration step in the animation of the particles, which approximates these influencing factors.

Section 2.3: Fluid Simulator Theory

A real time fluid simulation is a modeling device and there are multiple types of simulators. During the experimentation a grid-based simulator was developed. This simulator tracks cells in a cuboid lattice in space, every cell is tracked and updated every frame. The cells represent the state of the fluid's density and velocity for the region the cell exists in.

The particle system was animated with standard Newtonian physics with the acceleration climbing upwards and with arbitrary velocities added for a random effect.
It was quickly apparent that particles had to move relative to each other and a cheap model would not produce a desired realistic effect. Simply using layered textures did not give realistic movement. Advanced effects are already in use in the film industry. A very realistic flame can be rendered into a clip or into a component of a scene. The

computational cost of producing these images is very large, a single frame can take hours of processing time. It is therefore not realistic to use those techniques in a game environment, as the game needs to produce a new frame in real time, typically at 60 Hz. However, In order to faithfully animate a particle, dynamic flow and direction is required.
There are real time systems for dynamic flow, and there are a number of fluid simulators available which are capable of producing realistic flame movements. To achieve these convincing movements a real time fluid model, proposed by Jos Stam in his paper <Real-Time Fluid Dynamics for Games> (Stam J. n.d.), is employed.

A fluid simulator, specifically a Eulerian grid space fluid simulator, is a large collection of data points organised in a n-dimensional lattice. These data points are the states of the fluid within the space that the data point represents. Flow direction as a velocity and the density value are the standard Jos Stam proposed values, but more can be calculated such as temperature or pressure. Below are the Navier-Stokes equations for velocity and density:

Velocity:

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

Density:

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

Figure 2*: Navier-Stokes equations for the velocity field (top) and for displacing density (below). Taken from; Stam J. n.d.*

As we are interested in tracking simple characteristics for a flame. Specifically the velocity and the density at each point in our grid. The density is not actually tied to particles with mass and volume but is assumed and virtual.
It is more a representation of the fluid's energy at different locations and the dissipation / diffusal of the density from more dense areas towards less dense areas. This is an appropriate substitute for modeling energy diffusal / transfer.
The velocity field is used to animate the particles through the volume of the fluid simulator. The particle which will be animated needs to exist within the bounds of the simulator. The cell which influences the particle is first identified, the particle is then translated by the amount calculated in the simulator's cell.

The density field can be used as an approximation of energy. Instead of thinking of the diffusion of particles with mass, high density areas can be thought of as having more

pressure, high pressure areas as having more heat and the lower pressure areas as having less heat. Density does not actually describe heat in a real physical sense, it is merely used here to do so, as a shortcut. The density step in the simulator does not influence the velocity field. In reality density does affect flow. As a high pressure region moves towards a low pressure one there is a movement of particles which influences flow. Moreover, heat in a real world system typically forces a lower density in the heated region, causing the heat to rise with convection.

Both of these features, pressure flow and convection currents, are not modeled in the simulator, but for the purposes of game visuals simulating temperature accurately is not necessary. Making the assumption that the characteristics of density diffusion are the same as heat diffusion serves the project well.

The principle of real natural flame light is used to colour the particles. To simplify the problem, heat was examined alone without taking into account the colours produced by different chemical elements. If the particle is in a high density area we give it a hot colour in the white and blue range, if the particle is in a lower energy we give it a red or yellow range of colour. To model this for different elements or fuels, a colour temperature table for different elements would need to be created. If this approach is used, the table could serve as a fast lookup during the animation stage.

Machine performance is an important bottleneck when it comes to developing a fluid simulator for a game. Performance is also the argument against using fluid simulators in most cases for video games. As stressed in the introduction, a game could benefit from using a fluid simulator if the game has a mechanical or narrative reason to use one for flames, this follows for any other type of fluid as well. Using a fluid simulator for purely aesthetic reasons is still impractical due the large overhead costs on the hardware. Throughout the experimental part of this research I built the fluid simulator for both the CPU and the GPU.

Additionally, the openCL library uses a different programming paradigm to iterate over large sets of data. Stam's algorithm is designed as a conventional CPU single threaded function, using for loops to iterate over the data set. This is useful to prototype solutions but is not ideal for machine performance. The approach that was used for calculating the fluid simulations is referred to as a Grid-Based Method (Ertekin B. 2015), where an n-dimensional grid is fixed in space and each cell in the grid has defined values for their location in space. Moreover, we're specifically dealing with a Eulerian Grid where our simulated particles are translated across the grid according to the values at specific locations in the grid.
The algorithm to iterate the grid is also step based, meaning for each unit of time step, the whole data set needs to be processed before commencing the next step. Each step processes every element in the data set.

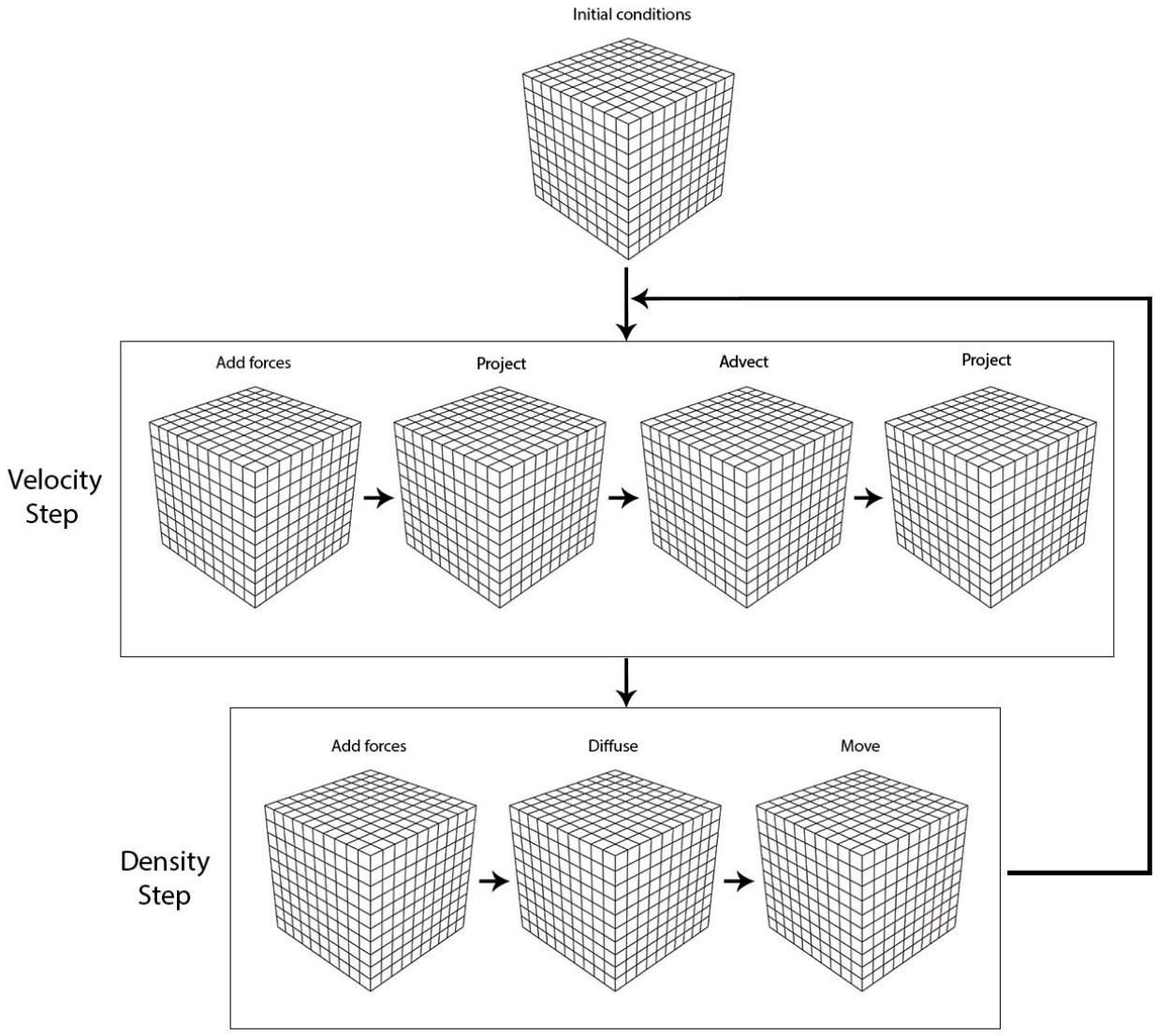

***Figure 3**: Rendition of Jos Stam fluid simulation routine order for every update*

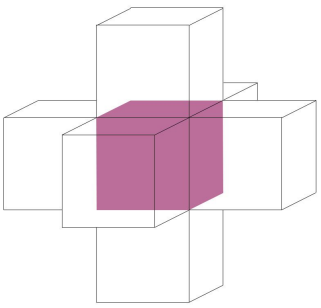

***Figure 4**: Diagram of the range of influence each cell in the fluid simulator need access to for computation*

The GPU device is composed of multiple hardware cells which can run a program. In openCL this is known as a program "kernel", this is essentially a program executed for each element in the set of data. The kernel is also considered the region each element is interested in. For a three dimensional fluid simulator we are typically interested in the 6 adjacent neighbouring cells to the cell we are computing. See Figure 4.

This explains the step based nature of the algorithm. Each step uses data from the neighbouring cells and therefore needs these cells to have up to date data to correctly crunch the new values for our cells.

OpenCL runs a kernel for each cell in the data set. The issue with this, as discussed before, is that the algorithm requires stages of precomputation before starting the next stages. We therefore need multiple openCL kernels per element, one kernel per stage of the algorithm per cell. In figure 3 we see the steps required for the fluid simulator during its update loop. We can assume that each lattice in the figure represents one kernel stage of the simulator, excluding the "initial conditions".
We can imagine that to calculate the overall fluid simulator, a chain of kernel events could be used: when the batch of "add forces" kernels has executed, the next batch of "project" kernels can be triggered and so on. However, this introduces many `wait()` calls or hardware pauses which is an undesirable feature when computing across large data sets.

Introducing these `wait()` calls is undesirable in parallel programming especially for large sets of data. This fundamental issue with the Stam algorithm only became apparent during the experimental phase of the research. Solutions for this problem are addressed later on in the document. However, they do not involve using Stam's paradigm.

This is not to say that fluid simulator's are not suited to running on the GPU, but that the Stam Grid-Based system is not well suited to running on the GPU. With some more research I now would suggest using a particle-based method (Ertekin B. 2015) for animating the particles as they are more suited to GPU bound calculations..
These methods are more suited to running on a GPU which would also allow for more particles to be simulated.

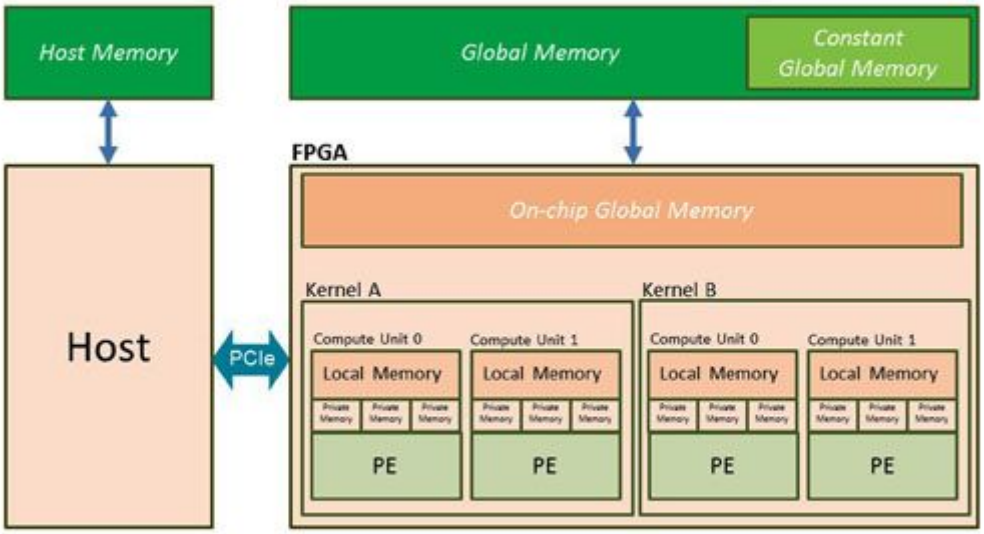

***Figure 5**: OpenCL memory structure with Field Programmable Gate Array (FPGA) hardware chips (Gilliland S. et al 2015)*

Section 3: Development

Octet is a custom c++ written open source engine. It provides access to openGL, openCL and bullet. It is also platform independent. This engine was used because it provides a good amount of placeholder constructs for openGL, making scene rendering straightforward and providing full control over the code base.

Parts of the research undertaken involved understanding the technical methods employed by the engine such as the resource management, shader to model structure, scene nodes renderer and many other industry techniques.

This was an insightful and interactive way to gain a better appreciation of how larger systems worked in code.

Octet being a small code base gave more control over the direct openGL and openCL calls. This is a very desirable feature for prototyping and testing ideas without the overhead of a large code base.

The majority of the coding work was completed using openGL and openCL. Octet provides openGL ready objects that perform the necessary calls for you. Some openGL features needed extensions to perform the task required. For example, the use of geometry shaders in the gl headers file had to be enabled as well as extending the code base to handle and complete geometry shaders.

OpenCL is also available in octet and is provided with a openCL class that wraps memories, kernels, devices and events. This required some changing for the experimentation work completed with openCL.

Section 3.1: Particle System

The first step in the development work was building the particle system. This was done mostly outside of version control as a prototype. Octet provides a number of examples, one of which uses a pre built `mesh_particle_system` object to render particles. The work on the prototype extended this class to add a few additional parameters to the particles, specifically a scale variable so that the particles can grow and shrink over their lifetime.

A colour variable was also used to track the changing alpha value of the particle. This is faded based on the lifetime of the particle. This is important because particle popping is very ugly and visible to the players of games. Using a smooth and rapid fade makes this look much better and feel more natural.

A simple smoothstep algorithm was implemented to map a value of lifetime to a variable of size and opacity.

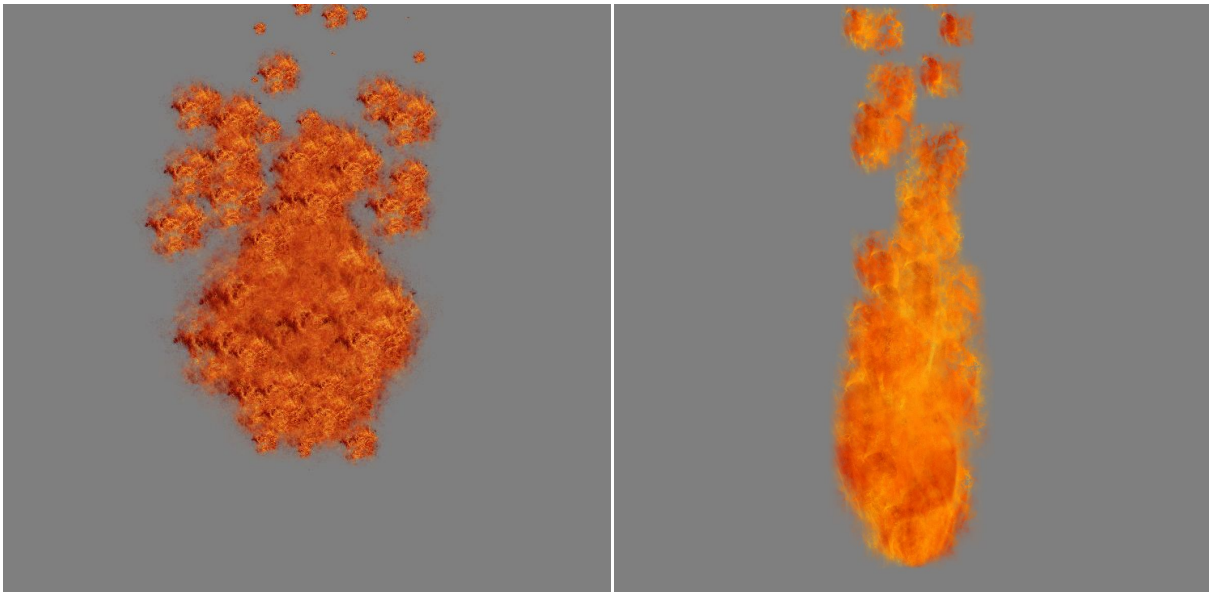

***Figure 6**: Image left, without overlay texture. Image right, with overlay texture, note the structure across sprites*

In order to implement opacity, the mesh vertex footprint also needed changing. A RGBA colour value was added to every vertex where the alpha is used with the output texture to render the particles fade in the fragment shader.

As well as this, more textures were added to be sampled from. Originally the aim was to add movement and coiling effects in the textures themselves. It was planned to pan along a seamless texture to obtain the sampled colour. This is an effect which was observed in the Unreal 4 engine, which uses Panner nodes (Epic Games 2015).
The base class renders one single texture. The work on the fire required more textures to be used, for example one which can be panned and one that acts as an alpha mask.
Another form of texture centric effect was also attempted. An overlay texture was used, the role of this texture being to blend together particles. The overlay is an imaginary texture that spans the overall area of the particle system. Particles throughout their lifetime move along this texture. In order to blend the overlay texture with the particle's mask and diffuse textures, the UVs of that particle are calculated by orthogonally projecting the vertices of the particle onto the plane of the overlay texture. This provides us with the UV values with which to correctly sample the overlay texture.
To make separate particles appear as part of a larger structure, the colour output of the diffuse particle texture is mixed with the diffuse overlay texture.

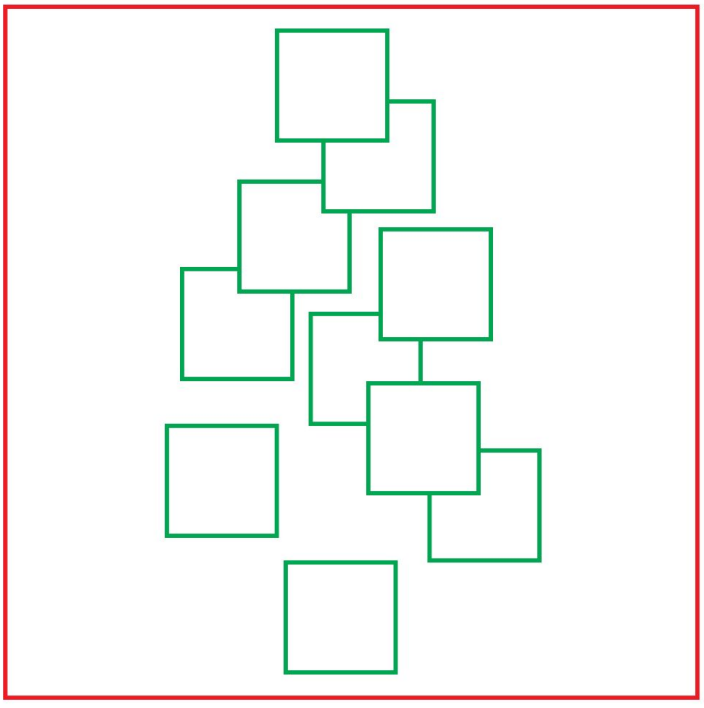

***Figure** 7: Green quads are particles, red quad is the overlay texture*

The existing `mesh_particle_system` that was used to build the `fire_particle_system` uses a dynamic array as a container for the particles. New particle objects are allocated memory and initialised then added to the array, as well as their particle animator objects. Particles that reach the end of their lifetime are removed from memory.

This is not an ideal structure to use as for every game loop a particle is added to the system and another particle expends its lifetime and is removed from memory. It would be far more efficient to remove the allocation and free calls and to use object pooling. When a particle is not being rendered and is therefore free, it is placed in a `free_particles` list. To add a new particle you just need to move the particle off the free list and reset its parameters. So it's lifetime is reset as well as its position, colour, rotation etc.
This system was not implemented for two reasons, the first being a lack of development time and the second being the requirement for some additional CPU overhead which is to be expected if this system is to be used in games. However, implementing this would be a high priority for release versions of the software.

Section 3.2: Fluid Simulator

Stam provides a thorough explanation and guide to write a fluid simulator based on the Navier Stokes fluid dynamic equations in his paper <Real-Time Fluid Dynamics for Games> (Stam J. n.d.).
Stam states that "In computer graphics and in games … what matters most is that the simulations both look convincing and are fast."
Work on the fluid simulator took up most of the development time. Writing it was straightforward as it was a well documented algorithm (Maguire B. 2012). The bulk of the research was to make this CPU bound algorithm run on the GPU. The GPU can be written to using hardware libraries such as openGL, openCL and many others all with specific purposes. These libraries are optimised to use purpose built hardware for performing rapid parallel arithmetic.

The particle system did not provide good enough visual effects. One issue was the animation of the system. This was due to unfamiliar movement in the particle system. For part of the research real flames were observed as well as video footage of real flames. One observation was that the larger a flames volume is, the more rapidly it moves and shifts.
A small candle flame does not move very much whereas a fireplace tends to whisp more and coil very regularly and rapidly. The chaotic coiling nature is referred to as turbulent flow in a fluid. The type of flow a candle flame typically demonstrates is known as laminar fluid flow. Laminar flow is a mostly stable system, where the velocities act in one structured direction, this is a desirable feature of a fluid simulator to test. More importantly this was a good building block to work from. With larger flames, laminar flow structures with regions of turbulence and eddy currents are expected to emerge.
The produced particle system alone did not provide this structure, therefore, a fluid

simulator was investigated to animate the particles instead of using the newtonian acceleration equations.
The resolution of a fluid simulator here is referred to as the number of cells, data points, within it. It should be expected that a higher resolution fluid simulation would be more suitable for larger flames. This is because a larger flame is more complex and detailed.

The implementation of a small fluid simulator capable of animating a candle flame was set as a preliminary milestone, before dealing with the larger and slower simulators.
Building the simulator was a straightforward process, Jos Stam wrote a very informative paper "Real-Time Fluid Dynamics for Games" which provides detailed explanations of how the simulator runs as well as code examples for a 2 dimensional simulator. An online example of a 3D Jos Stam fluid simulator also proved a very useful resource to build my own (Maguire B. 2012). This example code is designed for execution on the CPU. Later, GPU implementations were investigated.

The fluid simulator, calculates the values of density and velocity flow within a 3 dimensional cuboid. To achieve this, it makes use of the Navier Stokes equations. Jos Stam proposed a stable method which uses the direct neighbours of each cell as well as their previous states one step behind in time. This is done by calculating each cell which, when its value is calculated backwards in time, yields the same values we started with coupled with an iterative solver, specifically Gauss-Seidel relaxation.
The fluid simulator uses two steps; the `velocity_step` and the `density_step` are called during each loop. These functions are responsible for crunching the new values of the velocity field and the density field respectively.

The simulator also takes inputs directly in the data. Without any added forces it will eventually stabilise and return to an empty set of data.
A function was built that adds a force to the fluid simulator every loop. This is a small stable force to attempt to simulate the candle.
The force is injected into the simulator with a world coordinate of the position of the imagined reaction of fuel with energy and a vector of the magnitude and direction where the force is acting.
This is done by first intersecting the position of the force with the bounding volume of the fluid simulator. When there is an intersection, the corresponding cell in the fluid simulator is determined from the position. Then the velocity can be added to the simulator's velocity field.
A candle flame is simulated by inserting a single source point with a stable small force pointing upwards. This is used in the fluid simulator's add force step, every frame the force is added.

As well as needing to add force to the fluid simulator, values from it needed to be retrieved for the animation step. This is done using a similar technique to adding forces.

Based on the particle position it was determined where inside of the fluid simulator's bounds the particle is. Once the corresponding cell is found the velocity value is extracted and used to animate the particle.
There are some issues that were observed with performing the animation in this way, the fluid simulator does not necessarily have a predictable output; for two different simulator instances with different dimensions, the same force will not produce a similar amplitude of velocity values. For example, a very high resolution fluid simulator will have a smaller proportion of cells influenced by the input force than a low resolution fluid simulator which could have all of it's cells influenced by the same force direction and magnitude.

Finding a good input force to produce a good looking velocity output is trial and error and is different for any fluid simulator dimensions.
A small amount of time was spent looking into workarounds to standardise the forces and have them be proportional to the size of their respective fluid simulators. The work done is incomplete and for the immediate purposes unnecessary. However, in industry this is a very interesting problem.

There are two approaches that could solve this. The first is a simple proposal: maintaining a cell density for each fluid simulator. Each fluid simulator has a bounding volume. If we take the resolution of the fluid simulator as it's figurative "mass" and the volume the fluid simulator, we can find the density of cells as the number of cells / volume. As long as the developer maintains a constant density across all the simulators in the game then a standardised force system can be reliable.
The second method, needs to ensure that the distance in the scene that a force travels is preserved regardless of the resolution of the fluid simulator. The force needs travel through more cells for same amount of world distance in a dense simulator. The force can be adjusted so that it affects the right region.

The next focus was to implement the fluid simulator on the GPU to make use of the possible hardware acceleration. This is because the CPU could not handle the resolution required for larger flames. openCL was used, which is already supported by octet and was available with only minor code changes. Additionally, I used example code to learn the fundamentals of openCL in C (Apple Inc. n.d.).

A large effort was spent on researching and testing the openCL programming paradigm which is kernel based programing. OpenCL is a massively parallel programming environment which makes use of GPU arithmetic cores to crunch huge sets of numbers. The GPU hardware is composed of programmable chips and static routines implemented in silicon for optimal performance. The hardware used by openCL is the Field-Programmable Gate Array (FPGA), this provides the developer with a large array

of programmable cores designed to iterate through the indices of a data set in parallel. (See Figure 5)
An FPGA is well suited to the task we need to perform with the fluid simulator. However, there were complications with implementing the fluid simulator on the GPU. The kernel based programming approach is very well suited to crunching numbers provided all the data required is readable and available. The issue with the Jos Stam fluid model is its Eulerian Grid based approach. This requires steps to be fully executed before continuing onto the next step. Multithreaded programming is not well suited to this as many waits would occur until the whole data set has been processed for the next step. This dramatically slows down the calculation time and is not the correct approach in multithreaded programming.

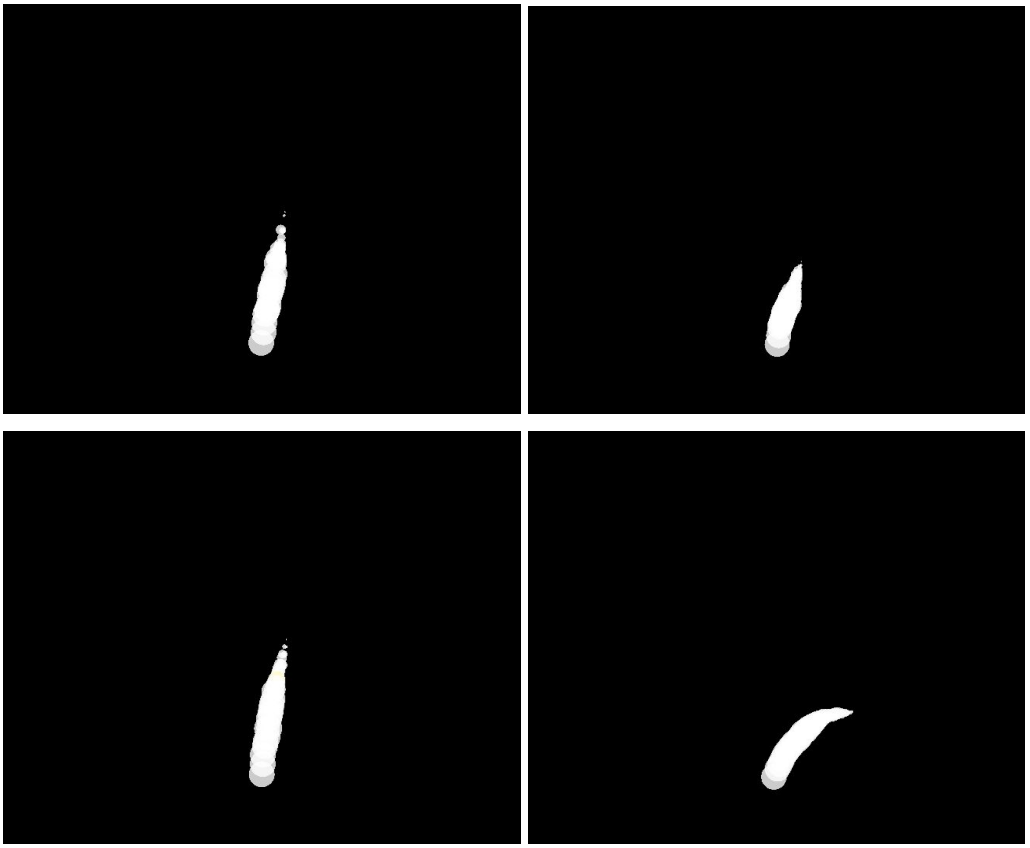

Figure 8*: Particles animated with fluid simulator's velocity field*

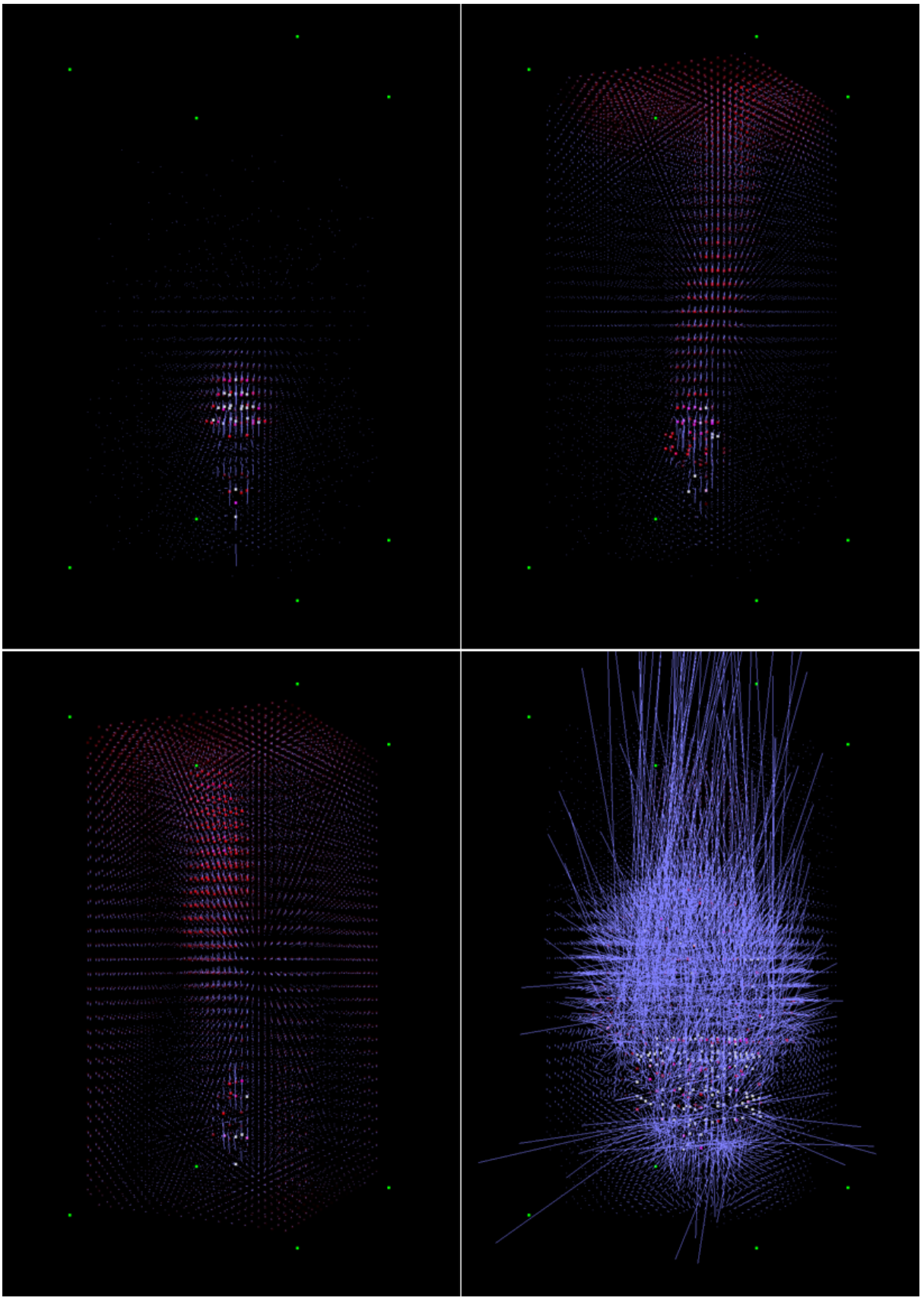

***Figure 9**: 3D visualised fluid simulator data. Blue lines represent the velocity field, Red and White represent low and high density cells respectively*

Section 3.3: Geometry Shader

The performance of the particle system required some optimisations to speed up the particles' update. The particle system originally taken from octet's `mesh_particle_system` object updates the particle mesh in the main loop every frame. This means that for each particle, four vertices are generated relative to the scale of the particle and the camera's normal. This is because, as billboard particles, these always need to face the camera. This is a substantial amount of computation for every particle on every game loop cycle. The solution proposed was to use a geometry shader to ease some of the calculations and offload them onto the GPU.

Octet does not support geometry shaders, geometry shaders were introduced with openGL 3.2 octet is built in this version as well. However, octet does not offer access to the GL literals related to geometry shader work. To fix this the gldefs.h header file simply needed modifying, the `GL_GEOMETRY_SHADER` token needed to be added. This token is used to define what type of shader is to be compiled and added to the render program.
Also, the `param_shader` object was extended with a `param_geom_shader` object which also adds a geometry shader along with the expected vertex and fragment shaders to the program.

Because a different rendering technique was used, some changes were required in the particle system class as well, these helped with overall system performance.
Instead of initialising the `fire_particle_system` as a `GL_TRIANGLES` render, `GL_POINTS` was used to reduce the amount of vertex updates required. With this geometry shader, it only needs to update the position of one vertex for each particle and the geometry shader extrapolates the four vertices used to render the particle quad. It does this using the projection matrix and the particle size to orient the quad to face the camera. Additionally, the footprint of the vertex can be reduced. The input we need is: position, scale and colour. This is 36 bytes per particle where before we tracked position, normal, uv and colour which is 192 bytes per particle (48 bytes per vertex), a dramatic memory footprint improvement.

I have not been successful with making the geometry shader render a quad in my octet fluid simulator environment. I believe that the process of enabling `GL_GEOMETRY_SHADER` token for openGL3.2 was not a complete enabling of this version of openGL. A necessary change to the gl defines files or library imports may have been omitted, more experimentation is required. This has something to do with the GLSL version syntax or with a missing call or link to enable the geometry stage. Despite my best efforts to fix the issue I've not been able to run the geometry shader in

octet although I can confirm the theory has been tested in a LWJGL (Light-Weight Java Game Library) rendering environment where I was able to test the geometry shader step.
More work needs to be done possibly on the underlying openGL version used by octet and in the octet gldefs.h file.

Section 4: Future Development

There are a number of ideas I would like to develop and test, these require more time to experiment with. These are a combination of features that were not fully fulfilled as well as different paradigms. These future tasks are provided with hindsight of the project and are a more likely technical solution to building a realistic flame.

Section 4.1: Particle-Centric Fluid Model

One possibility would be to use a GL compute shader to crunch the fluid simulator. With the fluid simulator in the GPU GL memory space the fluid data becomes accessible to the other GL shaders. The vertex shader could be used to animate the particles and determine their scale, lifetime, position and colour. A geometry shader could then be used to create a quad and generate UVs which would feed into a fragment shader to sample a fire texture.
With a system like this there risks being a large overhead on GPU memory and performance, however, this reduces the whole CPU routine down to two GPU calls per game loop, one compute call and one render call. Freeing up the CPU means the developer can focus on mechanics and narrative.

If asked to implement such a system for a game a different approach may produce more desirable results. Crucially, a particle-based fluid model would be used instead of the Eulerian grid method that was implemented. This would use particles relative to each other and does not calculate the values of a grid in the same way the eulerian grid based method used by Stam does.

For a fast implementation the ideal would be to run as much computation as possible on the graphics card. Because we are using a particle based fluid model the fluid calculation step can directly manipulate the particle position itself.

The particle system would be a mesh object of `GL_POINTS` each point representing a particle. In particle based fluid systems the particles are expected to carry their own descriptive properties; position, velocity, mass etc. The fluid simulator would use a known algorithm such a Smoothed Particle Hydrodynamics which is used in water fluid

simulations. These algorithms work to preserve a conservation of energy. The parameters of each particle could be stored in the mesh object within the vertex footprint.

The fluid simulation could be done in openGL using a compute shader to calculate the displacement of the particles . The compute shader can have access to the entire mesh object, the task here would be to write the new position to each particle before the render pass. The rendering order would therefore be; compute shader, vertex shader, geometry shader and finally the fragment shader.

As mentioned, the compute shader would update the new positions and velocities of the particles acting as an animation stage. This is because the compute shader, like the openCL kernels and the GL fragment shader, is executed on the FPGA. Then the next important stage would be to generate the quads for each particle in the system during the geometry stage. Then finally the fragment stage can take place with some texture sampling.

Using a method like this to render a flame keeps the entire flames calculation contained within the GPU which leaves the developer with more room for other game mechanics. However, this could introduce other complications. The fluid simulator is required to operate on a certain scene scale. Similarly to how the force input was standardised for all fluid simulators, the scale could need to be standardised for two flames to look similar to each other.

Section 4.2: Fuel modeling

At the start of this project my intentions were to design a flame based on fuel consumption tied in with realistic motion. The aim was to create a flame that was believably chaotic. The research explored the idea of the fuel source adding movement and colour variety as a part of the chaotic nature of the flame. The barrier between the energy and the fuel moves and shifts as the fuel is consumed, this adds natural movement to the flame.

The initial ideas on this topic were all about the fuel consumption, i.e. a moving flame source. Although these ideas were not addressed in the implementation of this research, there are some suggestions that can be useful guidelines.
One difficulty with modeling flames is to know where the flame should exist. As we know, a real flame's visible energy appears at the boundary of an oxygen supply and a fuel supply. The flame typically adds forces which influence the oxygen supply and sometimes the fuel supply, therefore the reagents in the reaction are dynamically moved by the flame. To model something as complex as this realistically is not

computationally viable for the output and for the most part would be invisible to an observer. We can simplify this model by assuming a stable flow of oxygen towards the reaction boundary. This means the frontier of the reaction always happens where the edge of the fuel is. The proposal is to track a fuel's volume in three dimensions.
The source of the flame is determined to exist on the surface of the fuel. This can be done using barycentric center of mass to identify a point on the surface of a fuel mesh.
This source point will be used as a location for the fluid simulator's `add_forces` step. As a fuel is consumed we expect the source point to move.
In a real flame system the source points are wherever the fuel is in the presence of enough energy. If there is enough energy the entire fuel source is engulfed. Moreover, in a real system the fuel has a combustion time and different fuels react in different ways.

In order to model the energy values and diffusal, a full simulation might require advanced volumetric modeling of pressure and temperature over time, there are techniques that can be used to achieve this. (Yngve G. *et al* n.d.)
However, similar heat and propagation techniques can be simpler. For instance, a simulated fuel would need a temperature map and a routine for fuel consumption. This could create interesting fuel spreading as when fuel is used up in one region the flame spreads to another region.

I propose a fuel mesh object, will have regular vertex data used to render the object in the scene. In addition to this the amount of combustible material could be stored as a float value per vertex. The remaining combustible material could determine the strength of the added force.
As well as a quantity of combustible material the fuel could also include a burn rate which would be representative of how flammable the fuel is. With a higher burn rate, more of the fuel is consumed in a single update of the fuel. On the fuel surface local temperatures will be hotter provoking more fuel sources to spawn which in principle in turn would increase the high temperature area of combustion. This would create an expected chain reaction and would add stronger forces to the fluid simulator to reflect that.

This could be coupled with a simple heat spreading model. For example; an exponential decay of heat around a point could determine an area in which the fire can spread. If there is a large enough fuel object, then a hot flame will have a larger potential spreading area than a colder flame. A heat threshold could be used to generate new flame sources, this would behave similarly to a flash heat temperature range. Some measure would be needed to ensure a minimum distance to other flame sources. This function would not need to be tied to frame rate either, a slower update could still provide a realistic spreading effect as long as we don't assume highly volatile fuels such

as gasoline which are consumed very rapidly and are more volatile fuels.

Section 5: Conclusion

Whilst working on this project I've explored a number of programming techniques and environments that I had not worked on before. I believe these have furthered and strengthened my experience and knowledge of low level engine programming which is an avenue I would like to continue to explore.

Firstly, working with Octet has been challenging but informative. The full range of access to the engine has required me to learn how to build on top of an already existing code base in a scalable manner. Moreover, I have had to work directly with new programming techniques and Object Oriented Patterns I had not encountered before, Such as the visitor pattern, inline library includes and practical applied hardware acceleration. Octet has been a very useful engine thanks to it's full source code access, it is a good programming sandbox.

Implementing an openCL program has been another personal milestone. This is my first use of the GPU for a non rendering task. From this experience I have learned the constructs of GPU programming as well as the memory and hardware design and the kernel paradigm's limitations. The skills used in executing software are translatable to other types of task such as ray tracers or physics pipelines using GPU hardware acceleration.
These new skills are very important to me as my current aspirations involve working close to game engines and hardware.

The fluid simulator represents the majority of the computation done during the project. A fluid simulator is a very expensive component to use in software. This raises the question, should a fluid simulator be used in games?

The answer depends on the game that is being designed of course but also on the role of the fluid simulator in the game. I can foresee the use of a fluid simulator in a game that makes use of the physical properties to add some engaging elements rather than just visual ones. The tomorrow Corporation designed a fire based game called "Little Inferno" in this game the player is only presented with a fireplace and objects to burn. It appears that the fire is calculated using some fluid dynamic equations. This game is a good example of how a focused and small scope leaves room for fluid dynamic calculation to occur. "In its final form the item burning elements of *Little Inferno* constitute around ninety per cent of the game." (Thubsticks 2014).

Moreover, Cities Skylines, the city simulator by Paradox Interactive, uses a fluid model for the rivers and seas in the game. Additionally, the boats use the Archimedes principle to determine how they float (ParadoxExtra 2014).
To answer the question, yes, fluid simulators should be used in games but the simulation should impact on the gameplay and not just be used to render visuals.

The aim of my research was to produce a final product of a realistic flame. I have not been able to create this. The work that was done demonstrates techniques and methodologies that do work towards this goal. Mistakes like using a Eulerian-Grid Based model were made because I had not yet learned about Particle-Based models.
With this Using a different design as mentioned in Section 4.1 I believe that a realistic complex flame could be rendered in real time.

Additionally, It is expected that over time hardware will advance to a degree where this is truly possible and my hope is that this work will become more relevant at that time. This paper aims to show how an algorithmically constructed flame could be designed.

With modern demands on hardware and software for new technologies such as computer vision, virtual reality and augmented reality. We can see that more resources will become available to developers. Additionally, in the case of virtual reality the need for convincing immersive realistic components will become increasingly relevant. Developers will become more familiar with hardware accelerated systems such as physics engines, ray tracers and indeed dynamic fluid models. Understanding how these systems work as well as experience of building them, is valuable and relevant for a modern developer.

Experimentation:

The experimentation work can be found on GitHub with the following link:

https://github.com/Deahgib/Fire-Opengl

The repository is structured with four branches. Each representing a specific set of work I've conducted. The gl32 branch represents the experimentation with geometry shaders. The openCL branch represents the work done with FPGA openCL programming. The initial commit is the work on the particle system. The refactor branch is the work done for the first supervised presentation.

Bibliography:

Thomason, A. (2016). *A single module, header only C++ framework for learning OpenGL.* Github [online] Available at: https://github.com/andy-thomason/octet [Accessed 17 Sep. 2017].

Khan Academy. (n.d.). *The quantum mechanical model of the atom.* [online] Available at: https://www.khanacademy.org/science/physics/quantum-physics/quantum-numbers-and-orbitals/a/the-quantum-mechanical-model-of-the-atom [Accessed 17 Sep. 2017].

Ertekin, B. (2015). *Fluid Simulation using Smoothed Particle Hydrodynamics.* MSc. Bournemouth University,. Available at: https://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc15/06Burak/BurakErtekinMScThesis.pdf. [Accessed 17 Sep. 2017].

Apple Inc. (n.d.). *OpenCL Hello World Example: hello.c.* [online] Available at: https://developer.apple.com/library/content/samplecode/OpenCL_Hello_World_Example/Listings/hello_c.html#//apple_ref/doc/uid/DTS40008187-hello_c-DontLinkElementID_4 [Accessed 17 Sep. 2017]

Epic Games (2015) *Animating UV Coordinates*. Unreal Engine [online] Available at: https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/HowTo/AnimatingUVCoords/ [Accessed 17 Sep. 2017]

Maguire, Blain (2012) *3dfluid/solver3d.c*. Github [online] Available at: https://github.com/BlainMaguire/3dfluid/blob/master/solver3d.c [Accessed 17 Sep. 2017]

Stam, J. (n.d.). *Real-Time Fluid Dynamics for Games.* Alias | wavefront. [online] Available at: http://www.intpowertechcorp.com/GDC03.pdf [Accessed 17 Sep. 2017]

Yngve, G., O'Brien, J. and Hodgins, J. (n.d.). *Animating Explosions.* Georgia Institute of Technology. [online] Available at: https://graphics.stanford.edu/courses/cs448-01-spring/papers/yngve.pdf [Accessed 17 Sep. 2017].

Thubsticks, (2014). *GDC 2014: Trial by Fire with Little Inferno*. [online] Available at: https://www.thumbsticks.com/gdc-2014-trial-fire-little-inferno/ [Accessed 17 Sep. 2017].

Gamedev, (2013) *Little Inferno - Fire Algorithm*. [Online Forum] Available at: https://www.gamedev.net/forums/topic/643713-little-inferno-fire-algorithm/ [Accessed 17 Sep. 2017].

ParadoxExtra, (2014). *Cities Skylines - Simulation Highlights.* YouTube. [video] Available at: https://www.youtube.com/watch?v=YrvybCBTQFE [Accessed 17 Sep. 2017].

Relevant reading material used during research:

Zhu, Y. (n.d.). *Fluid Simulation in Kernel Space*. [online] Available at: https://www.cs.ubc.ca/~mike323/doc/modelfluid.pdf [Accessed 17 Sep. 2017].

Enhua Wu, Youquan Liu , Xuehui Liu1 (n.d). *An Improved Study of Real-Time Fluid Simulation on GPU.* Faculty of Science and Technology, University of Macau. [online] Available at: http://imlab.chd.edu.cn/papers/CASA_2004.pdf [Accessed 17 Sep. 2017]

Braley, C. and Sandu, A. (2017). *Fluid Simulation For Computer Graphics: A Tutorial in Grid Based and Particle Based Methods.* Virginia Tech. [online] Available at: https://cg.informatik.uni-freiburg.de/intern/seminar/gridFluids_fluid-EulerParticle.pdf [Accessed 17 Sep. 2017]

Perlin, K. (1985). *An Image Synthesizer.* San Francisco, 19(3). [online] Available at: https://pdfs.semanticscholar.org/c4e6/00dad6313f9d0b5469ff0aa2f6e9f179de93.pdf [Accessed 17 Sep. 2017].

Xu, Y., Kim, E., Lee, K., Ki, J. and Lee, B. (2004). *Using PhysX Simulation Fire Model of Backdraft in Unity 3D Game Engine.* International Journal of Multimedia and Ubiquitous Engineering, [online] 9(6), pp.243-252. Available at: http://www.sersc.org/journals/IJMUE/vol9_no6_2014/24.pdf. [Accessed 17 Sep. 2017].

Reeves, W. (1983). *Particle Systems A Technique for Modeling a Class of Fuzzy Objects.* Computer Graphics, 17(3). [online] Available at: https://www.lri.fr/~mbl/ENS/IG2/devoir2/files/docs/fuzzyParticles.pdf [Accessed 17 Sep. 2017].

Perry, C. and Picard, R. (1994). *Synthesizing Flames and their Spreading.* Proc. of the Fifth Eurographics Workshop on Anim. and Sim., M.I.T Media Laboratory Perceptual Computing Section Technical Report No. 287. [online] Available at: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=1C5493AD50475968E96FF09241719BDD?doi=10.1.1.40.5005&rep=rep1&type=pdf [Accessed 17 Sep. 2017].

Lever, J. and Komura, T. (2012). *Real-time controllable fire using textured forces*. Vis Comput, 28, pp.691-700. [online] Available at: http://www.ipab.inf.ed.ac.uk/cgvu/fire.pdf [Accessed 17 Sep. 2017].

Lamorlette, A. and Foster, N. (n.d.). *Structural Modeling of Flames for a Production Environment*. PDI/DreamWorks. [online] Available at: https://ai2-s2-pdfs.s3.amazonaws.com/44eb/321ae694d20ad85c8fa7d79136716ada4a95.pdf [Accessed 17 Sep. 2017].

Fabrice Neyret, (2003) *Advected Textures*. Eurographics Symposium on Computer Animation, San diego, United States. Eurographics Association, pp.147-153, 2003,. [online] Available at: https://hal.inria.fr/inria-00537472/document [Accessed 17 Sep. 2017].

Gilliland, S., Vallina, F., Singh, V. and www.xilinx.com, X. (2015). *Compiling OpenCL to FPGAs* - Electronic Products. [online] Available at: https://www.electronicproducts.com/Digital_ICs/Standard_and_Programmable_Logic/Compiling_OpenCL_to_FPGAs.aspx?id=127 [Accessed 17 Sep. 2017].