# Topic:

Particle smoke and fluid simulation

Physics and Animation Coursework 2

Members:  
  **Baggio Pereira**  
 **Goncalo Lourenco**

Goldsmiths University of London

Computing Department

MSc in Computer Games and Entertainment

# Introduction

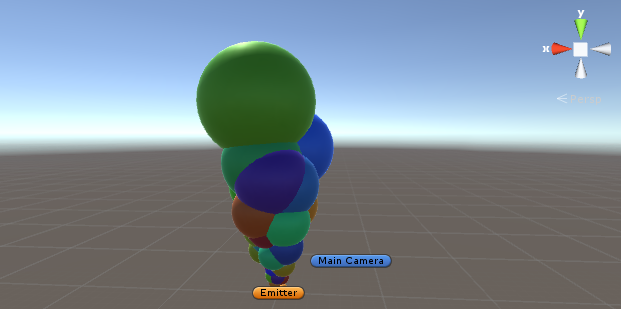
For our assignment we looked into creating a fluid dynamics system for smoke particles in 3D thinking about how nice it would look upon completion, however we did stumble upon the issue of creating the fluid dynamics for the smoke so we split the project in two. We created our own particle system to make a smoke effect and also looked into the creating a fluid dynamics system which is running on the GPU using shaders.

Our tool to create them was Unity since we have both used Unity heavily in our previous work.

Before we began creating the system, we looked into the background for each system and also into methods used before to create them.

## Particle system

Smoke effect was initially thought to work on GPU accessing it from a C# script. In the process of creating our own particle system in the hope to simplify Unity particles system and to make it more powerful on mobile applications. With that in mind start creating a basic particle Class on which was depending all particle management. Adding a collision detection with an effective repel system in which closest particle would go opposite direction on exception of Y axis (will explain it later). Just this two combined created an interesting dynamic which can be seen on video.



## Fluids system

The fluids dynamics system is a tool that is used to generate realistic animations of fluids such as smoke and water but not just including them. The system makes use of Euler equations or Navier-Stokes equations which describe the physics of fluids and can range from complex high quality animations used in movies to simple real-time particle systems that are used in modern games. There are a number of different approaches that can be used to create the simulation, the most common ones being a Eulerian grid method, smoothed particle hydrodynamics (SPH), vorticity method and Lattice Boltzmann method.

For our implementation we used a Eulerian grid, which is a cuboid space that is divided into cells where each cell contains the velocity and also other information such as density and temperature, that effect the fluid over time. Another way to implement the system is using a Lagrangian system, where particles are used and is better for expansive regions since they are not confined to a grid. Finally, there is a Hybrid system which use a combination of Eulerian grid and Lagrangian where particles that enter the grid will be affected by velocity of the grid, this type of system can be very useful for game worlds where Lagrangian can be used everywhere and a Eulerian grid can be used in very specific regions.

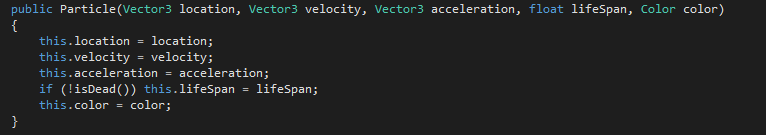
# Implementation

In this section we will be looking at the code and method of implementation of each system.

## Particle system

In the creation of a simple particle we need to take in consideration several variables:

1. Location, the position x, y, z of the variable in 3D space;
2. Velocity, speed of particle with direction defined because is a 3D vector;
3. Acceleration, gaining of speed as particle travels also in a 3D vector;
4. Lifespan, time that particle will be visible;
5. Color, just to give a distinction touch when rendering.



Beside these variables, particles can be also influenced by external factors like gravity and wind. For that purpose we got class Outside instantiated from Particle class.



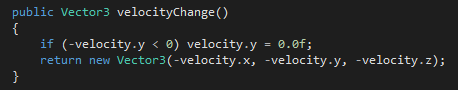
Particles creation process is based on random figures between a predetermined range with same point of origin.

Particles motion are constrained by two main factors:

1. By resetting acceleration value back to 1 when it reaches a higher value than 2;



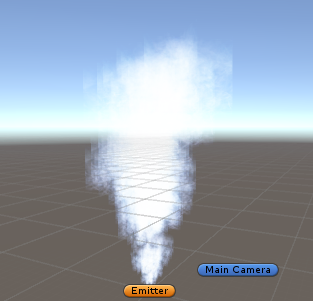
1. When collision occurs does not allow Y axis to have negative value.



These two options are the result of a highly experimental process and observation to simulate a certain desired behaviour, in this case smoke.

Closest particle is calculated by the square length vector of original point and remaining ones.

To finalize smoke texture was created using 4 flat planes with a transparency shader to obtain a nice look from every angle.



## Fluids system

As stated before, we used a Eulerian grid to generate a 2D fluid simulation, which incorporates the Navier-Stokes equations to describe the physics of the fluid. Looking into creating the fluid system on GPU we the chapter in GPU Gems that described the process in which the fluid calculations need to be done to recreate the simulation. Also we found a visual representation of the order of the methods needed along with code to help us recreate the system.

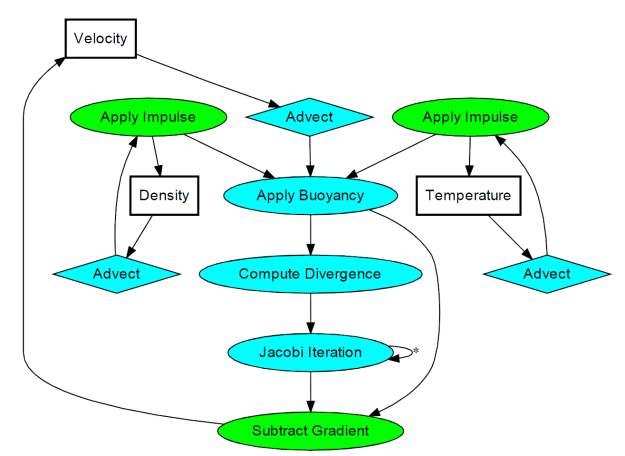


Fig.1 Visual representation of methods involved.

With this diagram we were able to get a start on creating each stage of the system.

### Advection

Advection is where the velocity of a fluid is used to transport object and densities along with the flow similar to adding dye to a flowing liquid and watching as the dye is being carried along with the liquid. In the system it involves the velocity of the cell along with density and temperature of the cell.

### Apply Impulse

The impulse stage is used to account for external forces, such as the user interaction e.g mouse movement to adjust the start location of the fluid.

### Apply Buoyancy

In our implementation, we use a temperature variable to make the fluid rise.

### Compute Divergence

In the divergence stage we find the neighbouring velocities and use their velocities to calculate the velocity that the cell should have. It also finds if there are any obstacles in its path which we will come to later.

### Jacobi Iteration

In this stage, it uses an iterative method to compute the precise pressure of the cell by getting the pressure of neighbouring cells. The number of iterations can be upwards of 50 but a good value can also be achieved from around 30-40 iterations.

### Subtract Gradient

In this stage, the pressure of the cell is calculated by getting the neighbouring cells again but its pressure is then subtracted by its velocity.

### Obstacles

This stage adds any obstacles to the grid space where it defines as that section of the grid to be empty so when the fluid comes near to this section, the fluid flows away from the section since there is no velocity, density or temperature in this area so the fluid has no force/information to use to calculate its flow. This stage is used in conjunction with the Compute Divergence, Jacobi Iteration and Subtract Gradient stages since these stages uses information from neighbouring cells.

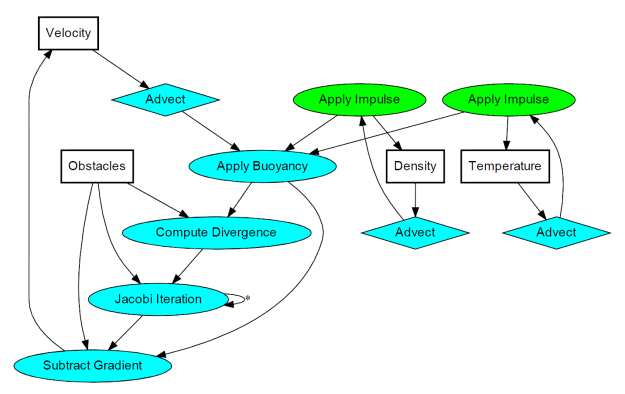


Fig 2. With obstacles stage added.

### The Code

We started off by having RenderTextures in Unity. These will be used as a sort of render buffers for the program. They don’t visually appear on screen unless assigned to a texture. They will hold all the velocities, temperature, pressure and densities. There are also RenderTextures that will have a combination of textures.Fig 3. RenderTextures used.

The velocity, density, temperature and pressure will have multiple RenderTextures which will be swapped around for reading and writing new data on it.

In void Start(), we create all the RenderTextures, giving them the format required for each RenderTexture . We also set the RenderTexture which is the final Texture that we will display on screen.

In void Update(), we call the different methods to calculate the fluid in the order as described. We start off with generating the obstacles and then the the advection stage by passing the velocity, temperature and density, separately and then swapping the textures around so the new calculated textures are used in later stages.

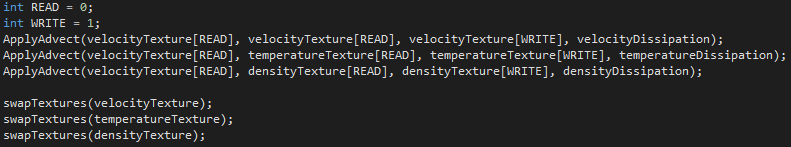


Fig 4. READ and WRITE are used to differentiate between the two RenderTextures.

We then calculate the buoyancy and impulse stages with the impulse stage taking the temperature and density separately and also so a swap of textures between each different stage.

Afterwards we calculate the divergence stage and start the Jacobi iteration, where the pressure RenderTextures are swapped after each iteration so that the data is always up to date and an accurate pressure can be calculated for each cell.

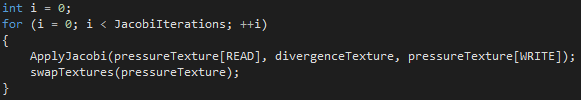


Fig 5.Jacobi stage.

Once the iterations are completed, it calculates the gradient stage before it the final texture is created to be rendered on screen.

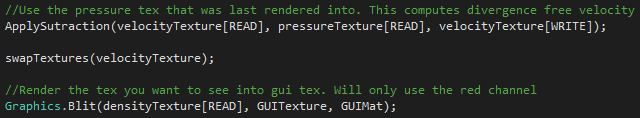


Fig 6. Final stage before rendered to screen.

Also in the Update() method, there is a method that is called upon to give the system some user interaction.

This method controls the fluid and obstacle positions, enabling and disabling the obstacles, and also increasing and decreasing the size of the fluid and obstacle.

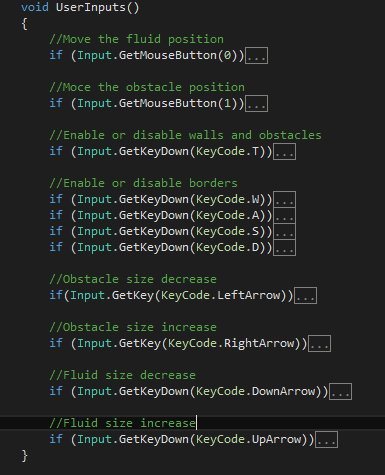


Fig 7. User controls

Since this system is using the GPU, all stages were written as shaders so they are run specifically on the GPU and will give us better performance.

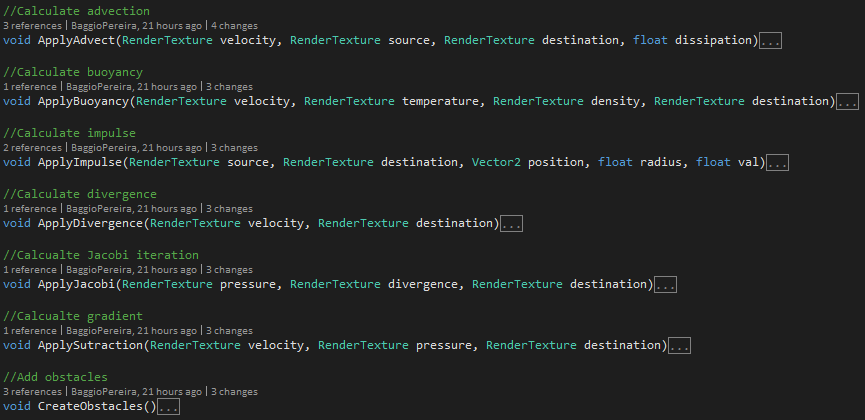


Fig 8. These methods are used to pass variables to the shaders for calculation.

## Results and Conclusion

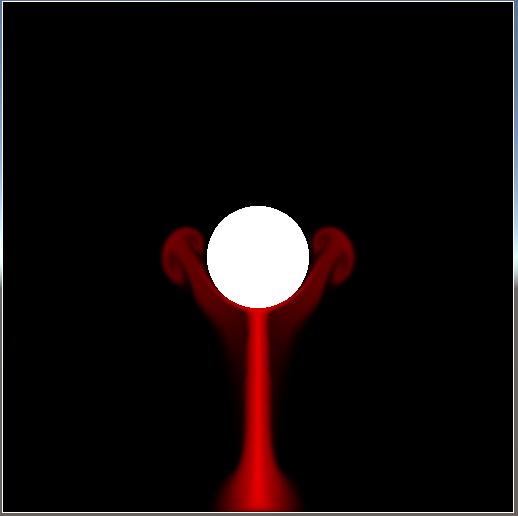
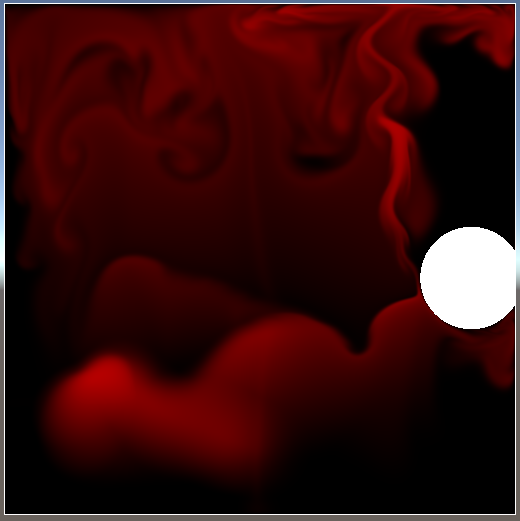


Fig 9. Final result



F 10. With moved fluid and obstacle

In conclusion, this was a fun and very interesting topic with a fantastic result. We did begin with creating a single shader for all the stages rather than having an individual shader for each stage. Hopefully with more time we manage to solve this issue and have a single shader to do all the calculations for the fluid system. Though we did have difficulties, we managed to solve each one quickly and there is plenty of examples which helped us along the way.

# References

*Fluid Simulation*. (2016, May 26). Retrieved from https://en.wikipedia.org/wiki/Fluid\_simulation

Harris, M. J. (2007, September). *Chapter 38. Fast Fluid Dynamics Simulation on the GPU*. Retrieved from nVidia Developer Zone: http://http.developer.nvidia.com/GPUGems/gpugems\_ch38.html

Rideout, P. (2010, November 21). *Simple Fluid Simulation*. Retrieved from The Little Grasshopper: http://prideout.net/blog/?p=58