GRID BASED FLUID SIMULATION USING UNITY COMPUTE SHADERS

Alex Christo

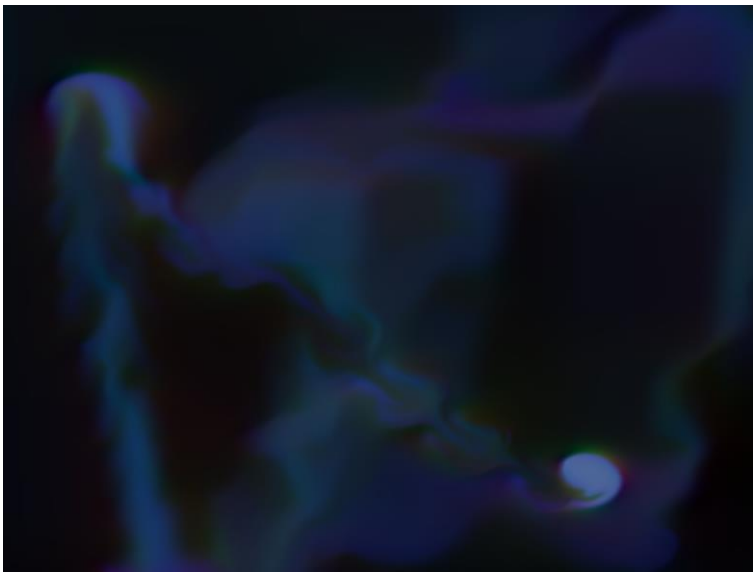

Figure 1: The Final Simulation in Action

1 INTODUCTION

Convincingly simulating fluids with computer graphics is a non-trivial task which has many different solutions, each with inherent benefits and drawbacks. This project focuses on creating a real-time, interactive 2d fluid simulation using the so called "Eulerian" or "grid-based" method.

2 MATHMATICAL BACKGROUND

Before exploring any specific simulation methods, it is prudent to first explain the Navier-Stokes Equations, from which they all derive. These equations work under the assumption that the density of fluid is constant in both time and space. This means if fluid leaves a space, it must be immediately replaced with fluid from elsewhere. This results in no change in volume in any one space (hence, it is incompressible). A fluid of this nature can be modelled using two vector fields, one for velocity and one for pressure. The velocity field describes the movement of the fluid and as such is the end result to solve for. Over time we describe the change in velocity using the aforementioned Navier Stokes equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{F},$$

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

Equation 1: the Navier Stokes equations

First let us break down the terms in the first equation. $-(\mathbf{u} \cdot \nabla)\, u$ represents advection, the transport of fluid caused by the current motion of the fluid body itself. $-1/\rho \nabla p$ represents pressure. When force is applied to the fluid it spreads through the volume, as the force acts on fluid molecules, pressure builds up causing movement. $\nu \nabla^2 u$ represents diffusion, depending on the viscosity of a fluid (ν) a fluid will be more or less resistant to flow. This resistance causes the velocity to "diffuse" throughout a fluid. Finally, F represents external forces such as gravity or wind etc. The second equation: $\nabla \cdot u = 0$, gives us the divergence of velocity vector. In a 2d case $\nabla \cdot u = \partial u / \partial x + \partial v / \partial y$. It measures the net change in velocity across a surface. For our fluid to be incompressible, this must = 0.

3 GRID METHOD

The originator of the grid method was Jos Stam in his 1999 paper **[1]**. Here a fluid is discretised into a grid (both 2d or 3d work the same), where values are placed at the centre of grid cells.

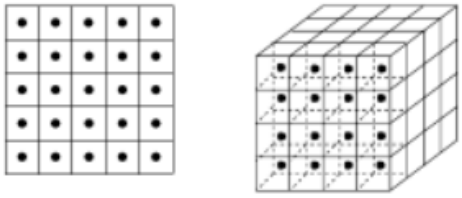

Figure 2: An example fluid grid [1]

The Navier Stokes equation for velocity (Equation 1) is solved in 4 steps. Starting from a resolved field we compute the field at a later time by: adding force, advecting, diffusing and finally projecting the solution into a divergent free field.

Adding force is trivial. Advection requires that we obtain the velocity of a point i,j at time t + δt. To do this we trace the point through the velocity field over time δt. We set the current velocity of i,j to be the velocity of the particle currently in that space, before the timestep multiplied by a dissipation constant (to ensure velocity and dye do not exist indefinitely):

$$\boldsymbol{U}^*_{i,j} = D * U_{[(i,j)-(U_{i,j^*}\,\delta t)]}$$

Equation 2: advection

We can reformulate the diffusion term into a Poisson equation by simplifying the Laplacian:

$$\boldsymbol{U}^*_{i,j} = \frac{U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} + ((\delta U)^2/v\delta t)U_{i,j}}{4 + (\delta U)^2/v\delta t}$$

Equation 3: Poisson form of $v\nabla^2\mathbf{u}$

This can be solved using an iterative solver (covered in the implementation section).

The final stage, projection, is the most involved. After advection, diffusion and force application the resultant velocity field (*W*) has non-zero divergence. Using Helmotz-Hodge decomposition, *W* can be decomposed into the form $W = U + \nabla P$,where *U* has zero divergence and ∇P is the gradient of a scalar pressure field. The newly created divergence can be corrected by subtracting the gradient of the pressure field:

$$U = W - \nabla P$$

Equation 4: divergence free field

We can calculate the pressure field by applying the divergence operator $\nabla\cdot$ to both sizes of this equation and letting $\nabla\cdot U = 0$ (from Navier Stokes). We are left with $\nabla^2 P = \nabla\cdot W$ which can be reformulated into the following Poisson equation:

$$\boldsymbol{P}^*_{i,j} = \frac{P_{i-1,j} + P_{i+1,j} + P_{i,j-1} + P_{i,j+1} - ((\delta U)^2\,\nabla\cdot W)}{4}$$

$$\text{Where } \nabla\cdot W_{i,j} = \frac{W_{i+1,j} - W_{i-1,j} + P_{i,j+1} - P_{i,j-1}}{2\delta W}$$

Equation 5: pressure field and divergence field

This Poisson equation can be solved using the same iterative solver method used for equation 3.

The gradient of this pressure field *P* is computed as follows:

$$\nabla P = \frac{(P_{i+1,j} - P_{i-1,j}), (P_{i,j+1} - P_{i,j-1})}{2\delta P}$$

Equation 6: pressure gradient

Finally, equation 4 can be solved to find the new, divergence free velocity field in the new time step.

4 ALTERNATIVE METHODS

The grid-based method is one of the two most popular general approaches to fluid simulation. Let us now consider its alternative, the "particle-based" method **[2]** so that we may compare it. Here a fluid is represented by some number of particles. Each particle holds a series of variables which represent its state. To briefly summarize, viscosity and pressure fields are derived from the Navier-Stokes equations similarly to the Euclidian method, with an additional term added to model surface tension effects. A method known as "Smoothed Particle Hydrodynamics" is used to loop over each particle finding its nearby neighbours. The density, pressure and acceleration is calculated for each particle and used to calculate the velocity and position of each particle in the new frame. The particle method does away with convection terms and any mass conservation equations (as a fixed number of particles ensures mass conservation). As such this method is usually faster than the grid-based method and ensures no loss of mass (which is hard to guarantee with the Euclidian method). However, grid-based methods have a much greater numerical accuracy due to using a fixed grid rather than unstructured particles. Given that compute shaders alleviated the speed issue, accuracy was a more important factor when selecting a method to implement for this project. Although, it is worth also mentioning that rendering in 3d is much simpler using the particle-based method, as particles can be directly used to render a convincing fluid body using methods such as marching cubes or point splatting. This ease of rendering and the fast speed means particle-based methods are particularly preferable for usage in video games.

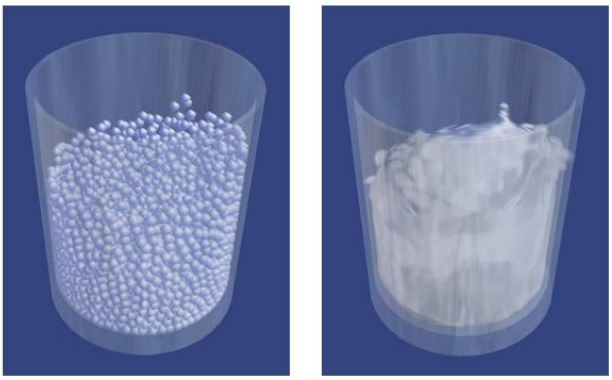

Figure 3: A particle fluid simulation [2]

An elaboration on the particle approach is presented by Park and Kim **[3]**. Here continually circulating flow is used as a basis velocity field, represented by vortex particles. Using a continuous swirling field guarantees no compression. The current state of vortex particles is represented by a vorticity field which evolves with each time step of the simulation. Combining these spinning fields results in a swirling motion that is especially well suited to representing gaseous fluids, but less applicable to modelling liquids.

Recently, research has been done to balance the trade-offs between grid and particle based methods. Golas et al. **[4]** present a hybrid approach, where a grid-based method is used to model the fluid surface, and particle based methods are used to model the fluid body beneath the surface. This way computation can be saved when computing the vast areas beneath the surface, but visual fidelity can be maintained on the visible surface. Clearly, this is still slower than a purely particle based method, but is ideal for large scenes, where both visual fidelity and performance are important factors.

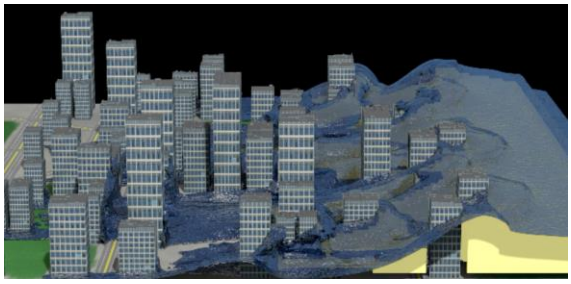

Figure 4: The hybrid method in action, here the yellow sections use a particle method, and the blue uses a grid based method [4]

Work has also been done to improve upon the grid based method presented by Stam**[1]**. Flynn et al. **[5]** attempt to address a similar issue to the hybrid method. They posit that a fixed resolution grid will either result in unnecessarily high amounts of detail where it is not needed, or vice versa. Instead of combining grid and particle methods, they suggest the use of a linear octree data structure to dynamically adapt the grid resolution, to add detail only where it is needed. This approach is much more adaptable than the hybrid method as greater resolution can be added or removed from anywhere in the grid. Additionally, linear octree stored contiguously in memory negates the inefficiency of previous recurse pointer based octree methods. This results in speeds up to 5 times faster than previous octree methods (and much greater than traditional grid methods), at the cost of additional memory usage.

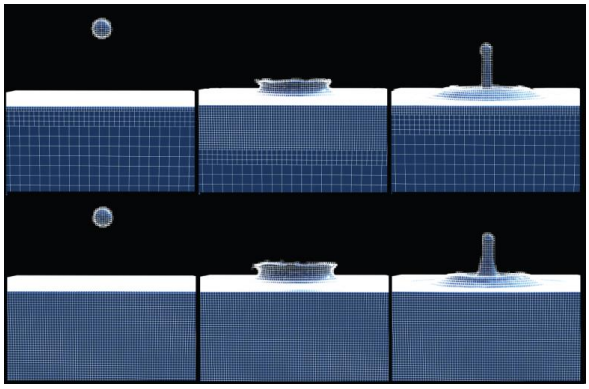

Figure 5: The octree method compared to a fixed grid [5]

Dong et al. provide a different way to improve grid-based methods, utilising neural networks **[6]**. Neural networks can be used to learn and model complex relationships based on input / output training data. Here they are used to approximate some of the costly calculations of the grid-based method. Further, this method presents the generation of several convolutional neural networks (CNNs), with different focuses regarding performance and simulation quality which can be switched out dynamically depending on the needs of the system. This dynamic CNN model greatly increases performance speed, but is heavily reliant on the breadth and accuracy of training data, and completely accurate behaviour cannot be guaranteed in all cases. This does somewhat diminish the numerical accuracy of the Euclidian method, which is one of its key strengths.

5 IMPLIMENTATION

The implementation method for this project is a 2d fluid simulation in the Unity engine. Unity was selected due to its support for compute shaders which were instrumental in increasing the computation speed of the simulation. The method implemented is based on that of the original Stam**[1]**, this was later reworked slightly to be directly suited to a parallel compute shader solution by Nvidea **[7]** as well as technical artsist Shahriar Shahrabi **[8]**. Both served as points of reference for the implementation created here.

To make the implementation as easy to follow as possible we will start from the highest levels and drill down. The main code body is a C# script which calls another C# script responsible for the actual fluid simulation logic. First an initializer method is called, this starts by instantiating a series of compute buffers. These compute buffers are used by an OSL shader to hold the grid data used throughout the fluid algorithm (notably the

velocity, divergence and pressure fields discussed earlier in theory). Additionally, an ink buffer exists to allow the fluid to be visualized onscreen, this can be thought of similarly to the velocity buffer, but it holds colour information instead of a velocity vector. After assigning values to some other parameters, kernel indexes are found. A kernel is essentially a block of code in the shader which can be called from the C# script via its associated index.

A destructor exists to release the buffers once the program is exited, this ensures that memory mismanagement does not occur.

Upon each update from the main C# script a tick function is called, this loop is where the fluid algorithm takes place. First the new mouse position is recorded (with some manipulation to record its position in grid space). First the advection method is called on both the ink buffer and the velocity buffer (with different dissipation factors for each), next diffusion on both, boundary checking for both, user force and ink are added into the scene and finally both buffers are projected. It is arguable that applying the projection stage to the ink buffer is not actually necessary, however the visual effect it gives it quite appealing.

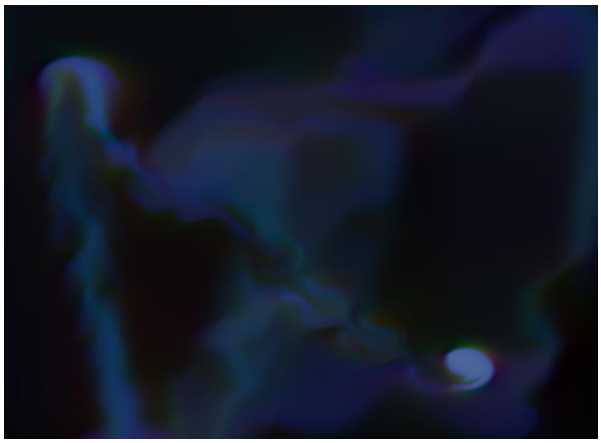

Figure 6: Simulation with ink projection

Figure 7: Simulation without ink projection

Before delving into each of these methods, we will explore how the compute kernels are called generally. A “DispatchKernel” method exists which takes a kernel index and an X ,Y and Z thread coordinate. The thread dimensions are directly tied to the dimensions of the simulation grid for simplicity. By default, the grid resolution is set to 672 by 672 as this was the highest possible at steady speeds on the machine of implementation and is 42 * Unity’s default 16,16,1 group size. This 16,16,1 simply means that on the GPU each control unit (CU) has access to 256 (16*16) threads. It is arranged in 3 dimensions simply to make access easier. So, each kernel operates on a 16 * 16 square of the simulation grid. The Dispatch kernel method divides the thread X,Y and Z it is passed by this group size. For example 672 / 16, 672 /16, 1 /1 = 42, 42, 1. So the simulation grid is actually split into a 42 x 42 grid on which a single kernel can operate on one segment (in reality multiple will likely be working in parallel to speed things even further). Once the correct location in this 42 * 42 grid has been computed it is sent to the shader via a dispatch command so that it knows which area of the grid to operate on. This allows many simple calculations to happen in parallel on different data due to the bank of threads all being controlled by a single CU, which in turn makes the fluid simulation much faster than if it was executed in a strictly linear fashion. As

much of the computation as possible is done via compute shaders for this reason.

The advection method works exactly as described in theory previously, tracing the point through the velocity field back in time to find its new value. This simulates the movement of the fluid based on its current motion. This is perfect for GPU implementation as the operation is what's known as a gather operation, meaning each thread can be mapped to a cell and only needs to write to the memory of this cell, while reading from neighbours.

The diffuse method has some added complexity, equation 3 described previously has only one known and 5 unknowns. Solving this traditionally is problematic. However, a solver can be used to iteratively get closer and closer to the correct answer. The one implemented here is a Jacobi solver. Initially, the values of the unknowns are guessed, and the result of the equations is computed. These results are then passed in as the values for the next iteration. As this process is repeated, we converge to a value very close to the truth. The more iterations, the closer the approximation will be. After some trial and error 100 iterations was chosen as it appeared to give accurate behaviour without slowdown (although lower values also work well). When iterating through the buffer used for the answer alternates between a temporary buffer and diffusion buffer to prevent a race condition where the same buffer is accessed by two threads at once. The number of iterations must be kept to an even number to ensure the correct buffer is written to at the end. This could have been alleviated by copying the buffer across in the final iteration, but the buffer copy was avoided where possible to increase speed.

An edgeCheck method ensures that the field values on edges of the grid are set to 0, preventing unwanted behaviour (this is known as a "no slip" condition).

The Ink method then adds ink to the ink buffer at the mouse position, in a circle which fades towards the edges.

The UserForce method adds velocity to the velocity buffer constantly to any pixels between the current and old mouse position. The method by which these pixels are found is taken from a 2d drawing utility by Shahriar Shahrabi.

The final step before rendering to the screen is projection, this has three stages as previously discussed: calculate the divergence field, use this to calculate pressure and finally use pressure to calculate a divergence free field. This is a direct implementation of the equations described in section 2. The same Jacobi method used for diffusion is again used here to solve equation 5.

Finally, the ink buffer is converted to a texture which can be blit to the camera.

6 SUMMARY + REFLECTION

To summarize, in this project we have explored various methods of fluid simulation, evaluating their benefits and drawbacks. A grid-based method was selected and implemented in Unity using compute shaders to speed performance. Overall, the visual fidelity and speed of the simulation are more than satisfactory. Use of compute shaders definitely provided a significant speed boost.

This project was the researchers first experience with compute shaders and implementing took longer than expected. As such there are some planned features that were not implemented due to time constraints. Implementing an octree structure as described in **[5]** would have boosted speeds even further. Additionally, the simulation is limited to 2 dimensions. The grid method makes extending to and rendering in 3d nontrivial, but a reasonable facsimile could have been achieved by projecting the simulation onto a flat plane in a 3d scene and using the pressure buffer to displace the plane up or down, giving the illusion of a 3d simulation **[8]**.

REFRENCES

[1] Stam, J., 1999. Stable fluids. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99,*.

[2] *Matthias Müller, David Charypar, and Markus Gross. 2003. Particle-based fluid simulation for interactive applications. In Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation (SCA '03). Eurographics Association, Goslar, DEU, 154–159.*

[3] *Sang Il Park and Myoung Jun Kim. 2005. Vortex fluid for gaseous phenomena. In Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation (SCA '05). Association for Computing Machinery, New York, NY, USA, 261–270. https://doi.org/10.1145/1073368.1073406*

[4] *Abhinav Golas, Rahul Narain, Jason Sewall, Pavel Krajcevski, and Ming Lin. 2012. Efficient large-scale hybrid fluid simulation. In ACM SIGGRAPH 2012 Talks (SIGGRAPH '12). Association for Computing Machinery, New York, NY, USA, Article 44, 1. https://doi.org/10.1145/2343045.2343104*

[5] Sean Flynn, Parris Egbert, Seth Holladay, and Jeremy Oborn. 2018. Adaptive Fluid Simulation Using a Linear Octree Structure. In Proceedings of Computer Graphics International 2018 (CGI 2018). Association for Computing Machinery, New York, NY, USA, 217–222. https://doi.org/10.1145/3208159.3208196

[6] Wenqian Dong, Jie Liu, Zhen Xie, and Dong Li. 2019. Adaptive neural network-based approximation to accelerate eulerian fluid simulation. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19). Association for Computing Machinery, New York, NY, USA, Article 7, 1–22. https://doi.org/10.1145/3295500.3356147

[7] Fernando, R., 2004. *GPU gems*. Boston [etc.]: Addison-Wesley, p.Chapter 38.

[8] Shahrabi, S., 2022. *Gentle Introduction to Realtime Fluid Simulation for Programmers and Technical Artists*. [online] Medium. Available at: <https://shahriyarshahrabi.medium.com/gentle-introduction-to-fluid-simulation-for-programmers-and-technical-artists-7c0045c40bac> [Accessed 30 May 2022].

[9] Shahrabi, S., 2022. *Compute-Shaders-Fluid-Dynamic-/2DDrawingUtility.cginc at main · IRCSS/Compute-Shaders-Fluid-Dynamic-*. [online] GitHub. Available at: <https://github.com/IRCSS/Compute-Shaders-Fluid-Dynamic-/blob/main/Assets/Shaders/Resources/FluidDynamic/2DDrawingUtility.cginc> [Accessed 30 May 2022].