UNIVERSITÉ Concordia UNIVERSITY

SPH simulations on the GPU

COMP 6311: Computer Animation

Pablo Arevalo Escobar
40081955

1. Introduction

Fluid simulations have long been an area of interest in the field of computer science, some of the principal applications include engineering, games, and movies. The field of fluid simulations can be segmented into three subsets, the Lagrange approach, the Eulerian approach, and the Hybrid approach. Each approach employs the Navier Stokes equations to define the dynamics of their system. The chosen approach depends largely on the desired result, does the simulation have to be accurate? Or does it simply have to be visually 'close enough'?

Smooth particle hydrodynamics (SPH) is a particle based, Lagrangian, approach to fluid simulations. The Lagrangian perspective discretizes the fluid system as a set of 'particles', each storing their own state and distributed freely without any specific structure. To understand SPH an understanding of the Navier Stokes equations is required.

2. Navier Stokes Equations

The Navier Stokes is the heart and soul of fluid simulations, its formulation is defined by two differential equations: the momentum equation and the incompressibility condition.

2.1 The Momentum Equation

This equation is defined by Newton's second law of motion, $F = ma$. The F is then replaced by the forces acting on a fluid: gravity, pressure, and viscosity. The acceleration is replaced by the material derivative of the velocity. This leaves us with the following formulation of the second law:

$$\frac{\partial u}{\partial t} = g - \frac{V}{m}\,\nabla \mathrm{p} + \frac{\mathrm{V}}{\mathrm{m}}\,\mu\nabla.\nabla\vec{u} - \vec{u}.\nabla \mathrm{u}$$

Note that a supplemental section is appended at the end which goes into more detail regarding the derivation of the momentum equation.

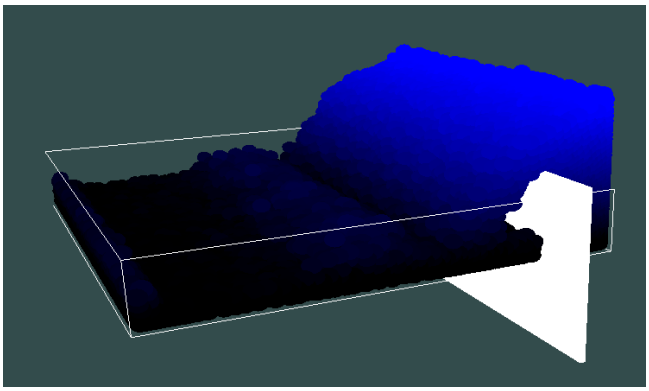

Figure 1. Still of the SPH Dam Break Scene

2.2 The Incompressibility Condition

The incompressibility condition comes from the assumption that the fluid we are simulating is incompressible and thus states that the density throughout the fluid is preserved, mathematically this is formulated as: $\nabla.\,\mathrm{u} = 0$.

3 The Lagrangian View

In the Lagrangian view, because of the discretized nature, we can remove some of the terms that came from the material derivative.

$$\frac{\partial u}{\partial t} + \frac{1}{rho}\nabla \mathrm{p} = g + \mu\nabla.\nabla\vec{u}$$

Where rho represents the density of the discretized volume of fluid. Because the system is discretized, a smoothing kernel is used to create a continuous field in which we can make use of the differential operators.

3.1 The Smoothing Kernel

The smoothing kernel, W, takes in a particle position and the position of its respective neighbors. It makes use of a predefined smoothing radius, *h*, and is a function which fades out to zero as the distance from the center approaches h. For a quantity q, we can compute its value at particle p with neighbors p_j with the following formulation:

$$q(x) = m\sum_j \frac{q_j}{rho_j} W(x - x_j) \qquad [1]$$

3. Implementation

This implementation of SPH consists of six core steps, each implemented through a compute shader in OpenGL. The first three steps are concerned with computing the nearest neighbours of a particle, the fourth step involves computing the density and pressure at each particle. The fifth step computes the internal forces acting on each particle and the last step performs time integration and computes the external forces acting on the system.

3.1 Nearest Neighbor on the GPU

The nearest neighbor algorithm involves computing all points that are within the smoothing radius, *h*, of a given point. Traditionally, this computation is done using a uniform grid that splits the space into a set of cells each of length *h.* The advantage of the grid formulation is that it provides a construction complexity of O(n) and a look up complexity of O (1).

It works by taking the position of a particle, hashing the value (which gives us the cell at its position), and binning it in a list which contains all the particles in that cell. Then when we make use of the smoothing kernel we can use this grid to have O (1) access to the cells that lie within the radius of the position. The implementation of this uniform grid becomes a lot more involved when performed on the GPU, however, the performance trade-off is worth the effort. This process is split into three steps: cell partitioning, cell allocation, and cell binning.

3.1.1 Cell Partitioning

The cell partitioning algorithm is run in a compute shader and is performed for each particle in the system. A buffer, gridCellCount, is passed into the compute shader along with a buffer, Pos, containing the positions of each particle. The particle position is then hashed to retrieve the cell index and an atomic add operation is used to increment the value at the index specified. The atomic add is used to prevent synchronization issues as the algorithm is ran in parallel in the GPU. At the end of this stage, each cell contains an integer that represents the number of particles stored in the respective cell.

3.1.2 Cell Allocation

The cell allocation algorithm is run in a compute shader and is performed for each grid cell in the system. The gridCellCount buffer from the previous stepped and a new cell Offsets buffer is passed into the shader. An atomic counter, CELL_OFFSET_VALUE, is also passed into the shader. This counter keeps track of the respective offset for each cell. For each cell in the grid, if the cell is empty then the loop ends. If not, the gridCellCount value is added to the atomic counter and stored in the cell offset buffer, the cell count value is then reset to zero. By the end, this cell offset buffer will store the respective offset value for each cell in the final array.

3.1.3 Cell Binning

The cell binning algorithm is run in a compute shader for each particle in the system. The now reset gridCellCount buffer, the cellOffset buffer, and a new particleBucket buffer is passed. For each particle, we once again compute the hashed cell index, we then retrieve the cell count at this index using atomic add. This will give us the current value of the cell at cell count (initially zero as we reset the array) and then increment the value by one. The particle index is then stored in the particleBucket buffer at the index specified by the offset of the cell + the cell count retrieved. At the end, we have a particleBucket buffer which can be indexed using the cellOffset and cellCount buffer to retrieve all the values present in a cell.

3.2 Density Pressure Computations

Because the force computations depend on density and pressure, the density and pressure values must be computed before running the force loop. This is implemented in a compute shader and makes use of the POLY6 kernel, which is standard across SPH implementations [1]. Computing the density at each particle is a simplified form of (1) and has the following form:

$$rho(x) = m \sum_j W(x - xj)$$

The density for each particle is then stored in a density buffer and the pressure is computed using the following formulation:

$$\mathrm{p}(x) = k(rho(x) - REST_{DENSITY})$$

3.3 Force Computation

The symmetric version of the kernel gradient is used for force computations, the symmetry means that the gradient calculated from two nearby particles with respect to each other is always the same. The computation for the pressure force used in the implementation is as follows:

$$f_p = -m^2 \sum_j (\frac{rho_i}{p_i^2} + \frac{rho_j}{p_j^2}) \nabla \mathrm{W}(\mathrm{x} - x_j) \qquad [1]$$

The spiky kernel, as used by Mueller [1], is employed in this case because the gradient of this kernel drops monotonically as the distance increases. If the POLY6 kernel was used instead, the pressure force will drop when particles are getting closer together. The viscosity force is computed as follows:

$$f_v = m \sum_j \left(\frac{u_j - u_i}{p_j}\right) \nabla^2 \mathrm{W}(\mathrm{x} - x_j) \qquad [1]$$

The two forces are then summed with the gravity force and stored in the force buffer

3.4 Time Integration and Boundary/Collision Handling

The time integration technique used is the explicit first order Euler:

Velocity[i] += DT*(force[i]/rho[i])

Position[i] += DT*Velocity[i]

Boundaries are handled by reflecting the velocity component along the axis of collision, the position of the particle is then reset to be just before the collision. Collision handling works by first checking the distance between the center of the particle and the object. If the distance is greater than the radius, then the closest axis (normal of the collision surface) is computed, and the force is resolved in the relative direction.

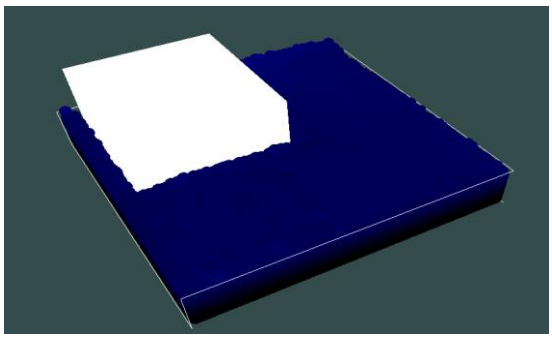

Figure 2. Collision Handling

4. OpenGL

The rendering is performed using billboards in OpenGL with an implicitly defined circle of radius 1. Billboards are a quad texture which is defined in the vertex shader (no passing of data between CPU and GPU), the modelView matrix is then altered so that the quad always faces the camera. In the fragment shader, all values which lie outside the radius of a unit circle are discarded. A color mapping that goes from blue to black depending on the particle height is then applied.

The number of threads in each of the compute shaders is always a multiple of 64, the optimal number of threads for Nvidia GPU's.

In addition to this post processing methods were attempted by loading the scene onto a framebuffer and using shaders to perform post processing on the scene.

Figure 3. Shows an attempt at post processing using a low pass filter (gaussian smoothing). Unfortunately, the performance hit was significant, and the program was unable to run reliably or stably. Because of the limited time it was decided to abandon this rendering technique as it would take too long to optimize.

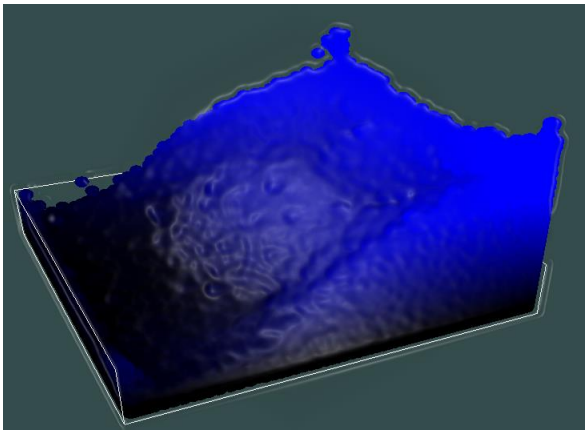

Figure 3. Gaussian Smoothing Applied in Post Processing

5. Accomplishments

The program can render, in real time, up to 160,000 particles (on a GTX 1070) due to the performance gains of the GPU. Boundary handling works without error and parameters, while extremely sensitive, can be tuned. Collision handling works well, and the UI can be used to manipulate simulator and object parameters. The UI can also be used to stop and start the scene at any given moment.

The fluid handles external collisions well with only a few edge cases. The "BOX EMERGE" scene can be used as a key example; in this scene a box is place below the boundary. Once the fluid has stabilized the user is expected to slowly raise the box so that it emerges from the fluid. The behavior in this scene is stable at the pre-set parameters.

6. Points of Improvements.

The system is highly unstable if the timestep is too large and therefore requires an extremely small timestep to behave as expected. The final timestep used for the project was 0.0007, this value should ideally be stable at a minimum of 0.016. Because of the systems unstable nature, the program is very sensitive to parameter changes and therefore there was a limited amount of parameter configurations that could be explored. Time management is also a key area of improvement, too much time was devoted to exploring the theoretical aspects of the fluid simulation and on premature optimization.

For example, a vector field function was developed to visualize vector fields to gain a better understanding of the Navier Stokes equations, this, however, proved to be completely unrelated to the functionality of the final implementation. Time was also wasted integrating the simulation into a rendering engine, Dune, which I've been developing in my spare time. In the end I decided to separate it from the engine and make it its own independent module. The lesson learned is to laser focus on the problem at hand and only once the core issues have been solved: stability and behavior of the simulation, to focus on external problems such as the rendering. A large amount of time was also wasted debugging the parameters and trying to find the source of the instability.

7. Conclusion

The project was able to successfully perform fluid simulations with simple collisions, as well as allow for the users to dynamically change the parameters. Given the time constraints, the project was an interesting, successful, and highly educational experience. Improvements, such as lighting and marching cubes rendering could significantly increase the quality of the project and would be interesting to implement in the future.

References:

[1] Kim, D. (2017). Fluid Engine Development (1st ed.). A K Peters/CRC Press. https://doi.org/10.1201/b22137

Libraries Used:

Shader class from: https://learnopengl.com/Getting-started/Shaders

Camera class from: https://learnopengl.com/Getting-started/Camera

GLFW – Window Functionality

GLM - Mathematical library for OpenGl graphics

Glad - OpenGl

ImGui – UI

SUPPLEMENT The Momentum Equation

This equation is defined by Newton's second law of motion, $F = ma$. The F is then replaced by the forces acting on a fluid: gravity, pressure, and viscosity. This leaves us with the following formulation of the second law:

gravity + pressure + viscosity = ma

Gravity is defined by *mg*. Pressure is defined by high pressure regions pushing into low pressure regions, however, only the net force acting on a particle is relevant. This imbalance can be measured by taking the negative gradient of the pressure, -∇p.

This can be understood as the negative gradient pointing away from regions of high pressure towards regions of low pressure. To get the force this value must be integrated over the region of the fluid, *V*. Defining the pressure force as: - V ∇p.

The internal viscosity of a fluid can be understood as the amount of resistance a particle faces when moving. This force tries to make P move at the average velocity of its neighbors. The Laplacian operator is used as it is a measure of the divergence of the gradient. We can then define the viscosity of a point P in the fluid as μ∇. ∇u. Where μ is defined as the 'dynamic viscosity coefficient' and can be manipulated to determine how viscous the fluid is. This force is also integrated over the volume of the fluid, *V.* We can then define the second law as

$$mg - V\,\nabla p + V\mu\nabla.\,\nabla u = ma$$

The acceleration term, *a*, can be obtained by taking the derivative of the velocity, however, because the velocity of a fluid varies in both space and time the material derivative is used. Where:

$$\frac{Du}{Dt} = \frac{\partial u}{\partial t} + \vec{u}.\nabla u$$

The momentum equation can then be seen in the whole as:

$$mg - V\,\nabla p + V\mu\nabla.\,\nabla u = m\left(\frac{\partial u}{\partial t} + \vec{u}.\nabla u\right)$$

which is often simplified to:

$$\frac{\partial u}{\partial t} = g - \frac{V}{m}\,\nabla p + \frac{V}{m}\,\mu\nabla.\,\nabla\vec{u} - \vec{u}.\nabla u$$