6.837 Final Project Writeup

Real-Time Smoothed-Particle Hydrodynamics

Mark Theng and Zhao Jingyi

1 Introduction

Real-time fluid simulation is traditionally performed using Eulerian (grid-based) methods, as they are easier to parallelize on the GPU than smoothed particle hydrodynamics (SPH). On the other hand, particle-based methods offers greater flexibility and interactivity, since many physical effects can be easily recast in terms of particles and force fields.

The most challenging part of implementing SPH on the GPU is collision force computation. For performance, only particle pairs sufficiently close to each other should be considered for collision. This requires collision detection, a task generally considered difficult for the GPU.

This project implements a complete real-time SPH simulation and rendering pipeline on the GPU using CUDA and OpenGL. On a NVIDIA GeForce GT 750M card, it handles 3000 particles at about 300fps – enough for the fluid to feel organic without showing signs of instability.

2 Background Work

A similar pipeline was previously implemented by Zhang[1], but for a quad-core CPU. Zhang reports that his implementation runs at 100fps for 3000 particles (including rendering), which our implementation surpasses.

Real-time SPH has been extensively studied by Pirovano[2]. His thesis implements real-time SPH with CUDA, but without screen-space rendering, achieving a run-time of about 3mspf (milliseconds per frame) for 4096 particles – similar to ours. Note that this is a very rough comparison, since their tests were run with different simulation parameters and on a different machine.

We also adapted from the uniform grid collision detection scheme described

[1] Zhang, J. (2016). Real-Time SPH Fluid Simulation. Retrieved from `https://drive.google.com/file/d/0B2macuhZeO18TzQ0UmNsRUt6bUE/view`

[2] Pirovano, M. (2011, December). Accurate Real-Time Fluid Dynamics using Smoothed-Particle Hydrodynamics and CUDA. Retrieved from `https://www.politesi.polimi.it/bitstream/10589/33101/1/2011_12_Pirovano.pdf`

$$\ddot{\vec{x}}_i = \vec{g} + \vec{F}_i^{(c)} + \vec{F}_i^{(v)}$$

$$W(\vec{x}) = \left(1 - \frac{|\vec{x}|^2}{R^2}\right)^3$$

$$\rho_i = \sum_j W(\Delta\vec{x}_{ij})$$

$$P_i = \frac{\rho_0}{\gamma}\left(\left(\frac{\rho_i}{\rho_0}\right)^\gamma - 1\right)$$

$$\vec{F}_i^{(c)} = F_0^{(c)} \sum_j \frac{P_i + P_j}{2\rho_j} \nabla W(\Delta\vec{x}_{ij})$$

$$\vec{F}_i^{(v)} = -F_0^{(v)} \sum_j \frac{\vec{v}_i - \vec{v}_j}{\rho_j}\left(1 - \frac{|\Delta\vec{x}_{ij}|}{R}\right)$$

Figure 1: The SPH equations. $\vec{g}$ is the gravitational force, $\vec{F}_i^{(c)}$ is the contact force, $\vec{F}_i^{(v)}$ is the viscous force, R is the particle radius, $\Delta\vec{x}_{ij} = \vec{x}_j - \vec{x}_i$ is the relative position between two particles, ρ_0 is the steady-state density of the fluid, γ is the Tait incompressibility factor and $F_0^{(c)}$ and $F_0^{(v)}$ are scaling constants.

by Le Grand[3] and the screen-space rendering scheme described by Green[4]. We used the CUDA Thrust toolkit for common GPGPU operations like sorting.

3 Methodology

3.1 SPH Equations

Particle positions and velocities are advanced using the Euler method with the SPH framework provided by Müller et. al.[5]. Viscosity is used instead of drag to maintain numerical stability. The Tait equation[6] is used to reduce compressibility, but with a lower incompressibility factor to improve stability. The slight compressibility of our fluid is mostly unnoticeable, since oscillatory effects are quickly damped out by viscosity. The equations used are provided in Figure 1.

When a particle with position $\vec{x}_i$ and pressure P_i collides with the boundary, a virtual particle at the projection of $\vec{x}_i$ to the boundary with density ρ_0 and

[3]Le Grand, S. (2007, August 12). Chapter 32: Broad-Phase Collision Detection with CUDA. Retrieved from `https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch32.html`

[4]Green, S. (2010). Screen Space Fluid Rendering for Games. Retrieved from `http://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf`

[5]Müller, M., Charypar, D. and Gross, M. (2003). Particle-Based Fluid Simulation for Interactive Applications. Symposium on Computer Animation, pp.154-159.

[6]Monaghan, J. (1994). Simulating Free Surface Flows with SPH. Journal of Computational Physics, 110(2), pp.399-406.

```
for each particle in parallel:
    let (x, y, z) be the particle's position, where
        each coordinate has p bits of precision
    let id be the particle's ID
    for each cell the particle touches:
        let h be a bit indicting if the cell
            contains the particle's center
        store CellTouch(
            hash = (x << (2p + 1)) +
                (y << (p + 1)) +
                (z << 1) + h,
            id = id
        ) in cellTouches
sort cellTouches by hash with parallel radix sort
for each cell touch in parallel:
    if, ignoring h, its hash differs from the next
        cell touch's hash:
        write current index into chunkBoundaries at
            the same index
apply parallel stream compaction on chunkBoundaries
for each chunk in parallel:
    for each cell touch in chunk:
        if h is set:
            assign chunk as particle's home chunk
for each particle in parallel:
    for each other particle in home chunk:
        if particles collide:
            add particle to collision list
```

Figure 2: Pseudocode for collision detection on the GPU.

pressure $\max(P_b, P_i)$ (where P_b is some constant) provides contact and viscous forces. This prevents low-pressure particles from sticking to boundaries and high-pressure particles from getting pressed against them. To allow particles to slide freely along the boundary, only the normal component of the viscous force is used. If a particle crosses the boundary, it is projected back to it with its velocity reflected as in an elastic collision.

3.2 Collision Detection

Our collision detection scheme is adapted from the uniform grid scheme described by Le Grand, shown in Figure 2. We use a grid with a cell size of $2R$, so that each particle contacts exactly 8 cells.

As we will see, the last step is the main performance bottleneck. This is because a large number of particle positions must be retrieved in a random-access manner for narrow-phase collision testing. This makes the CUDA kernel memory-bound. Using this insight, we achieved a small speedup by calculating

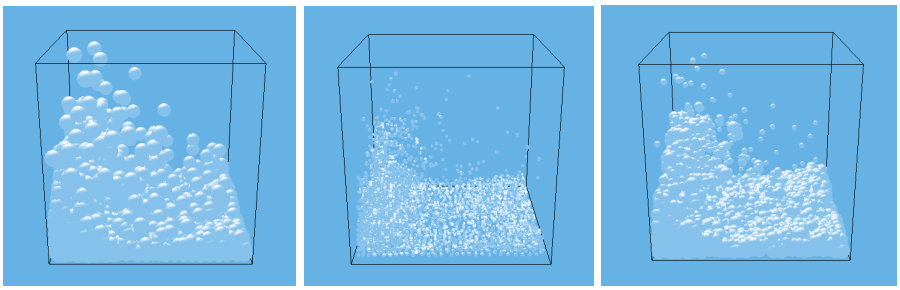

Figure 3: Spray particles should be small to minimize occlusion (left), while bulk particles should be large to provide a continuous surface for later passes (middle). Our implementation (right) uses $r_i = \text{clamp}\left(30\left(\rho_i^{0.2} - 1\right) + 0.8, 0, 2\right)$ (not physically based).

$$L = L_0 \text{mix}\left((1 - V \cdot H)^5, 1, R_0\right)\left(\max(0, N \cdot L)\right)^\gamma$$

Figure 4: Our lighting model. V and L are the view and light vectors (both pointing outward), N is the normal vector, $R = \text{reflect}(-L, N)$ is the reflected light vector, $H = \frac{V+L}{|V+L|}$ is the half-vector, R_0 is the Schlick reflection coefficient, γ is the shininess and L_0 is a scaling factor.

densities in the same kernel, reusing the retrieved neighboring particle positions.

3.3 Rendering Spheres

In the first pass, we render particles as billboard discs, modifying fragment depths to turn them into spheres. For aesthetic reasons, particle radii r_i are varied according to their densities as described in Figure 3. For lighting, we use Blinn-Phong specular reflection combined with Schlick's approximation for Fresnel reflection, as described in Figure 4.

Modifying fragment depths disables the hardware early depth test since the GPU cannot determine a fragment's depth before running the fragment shader. By positioning billboards a distance r_i closer to the camera, we can generate a lower bound for fragment depths independent of the fragment shader. This, along with the ARB_conservative_depth extension, allows for a limited form of early depth testing, improving rendering performance tremendously.

3.4 Smoothing

Following Green, we smoothen the fluid surface for a more realistic-looking fluid according to the pipeline shown in Figure 5. We use the bilateral filter (Figure 6) to preserve edges. Specifically, we apply the 1D bilateral filter to the depth map, first vertically, then horizontally.

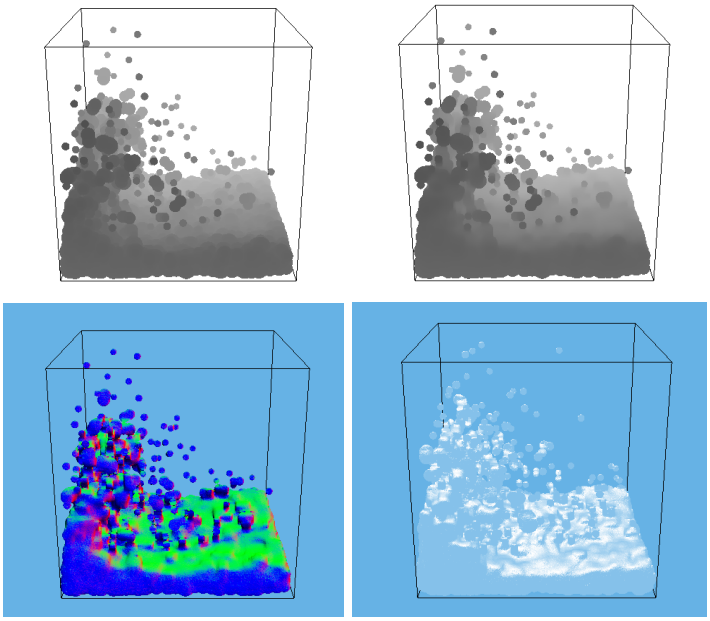

Figure 5: Our rendering pipeline. Initial depth map from sphere rendering (top left), smoothed depth map (top right), extracted normals (bottom left) and final output (bottom right).

$$I'(x) = \frac{\sum_{i=-M}^{M} I(x+i)w\left(i, I(x+i) - I(x)\right)}{\sum_{i=-M}^{M} w\left(i, I(x+i) - I(x)\right)} \text{ where } w(a,b) = e^{-\frac{a^2}{\sigma^2} - \frac{b^2}{\sigma'^2}} \quad (1)$$

Figure 6: The 1D bilateral filter. $I(x)$ is the depth map, M is the kernel size, and σ and σ' are smoothing parameters.

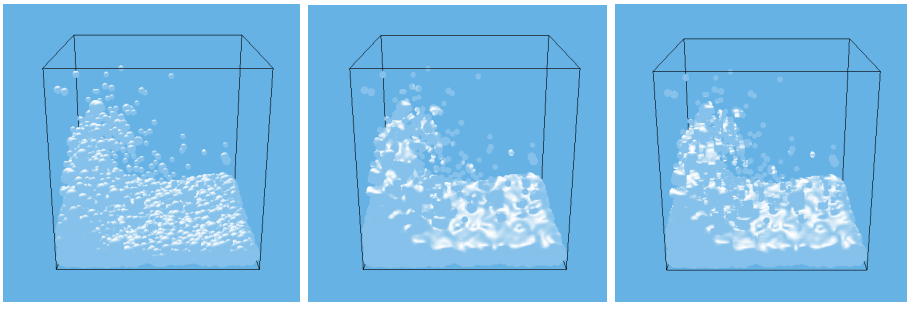

Figure 7: Comparison of the fluid without smoothing (left), smoothed with the 2D bilateral filter (middle), and smoothed with two 1D bilateral filters (right). Double-precision floating point was used for this comparison.

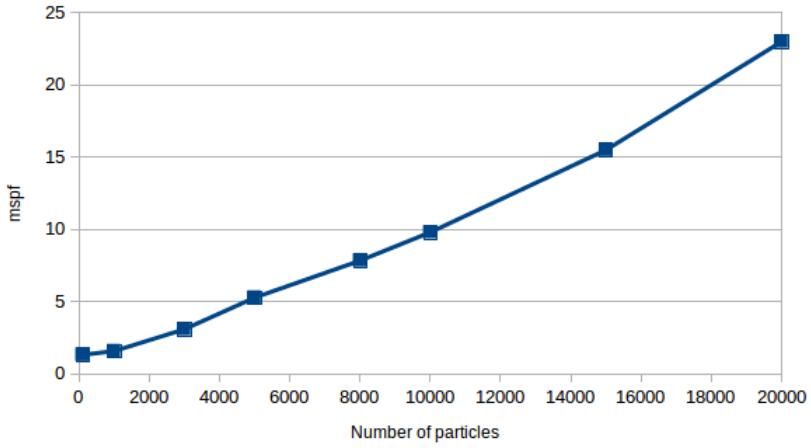

Figure 8: Performance of our pipeline as the number of particles grows, measured in milliseconds per frame (mspf). Measured by letting particles come to rest in a tall, fixed-size bounding box.

While this is not equivalent to the 2D bilateral filter (Figure 7), the performance gain is well worth the slight degradation in visual quality. We significantly reduce visual artifacts by making the depth smoothing parameter smaller for the horizontal pass than the vertical pass.

The normals are then extracted from the smoothed depth map, using an approximation of the fluid surface found by computing fragment positions in view space using the inverse perspective matrix.

4 Results

Figure 8 shows that our pipeline run-time scales well with the number of particles, which is enabled by the uniform grid collision detection data structure. Rendering performance depends on the camera position – when the camera is very near to the particles, the large number of fragments in the first pass dominates the frame rate.

In a test using a setup similar to previous figures, run-time was split 1.92 mspf for simulation and 1.11 mspf for rendering (including 0.52 mspf for the bilateral filter), with 0.39 mspf for other tasks like drawing the bounding box. Figure 9 provides a detailed breakdown for the simulation.

The attached live capture demonstrates 3000 particles simulated and rendered real-time on a NVIDIA GeForce GT 750M card, showing excellent stability despite fast-moving boundaries.

Computing collision lists and densities	50.55%
Sorting cell touches	19.16%
Computing collision forces	12.85%
Finding home chunks	9.10%
Computing particle densities	4.24%
Generating cell touches	3.25%
Compacting chunk boundaries	1.83%
Computing boundary forces	1.17%
Finding chunk boundaries	0.74%
Updating particle states	0.60%
Enforcing boundary conditions	0.40%
Computing external forces	0.22%

Figure 9: Relative time spent on each CUDA kernal as reported by nvprof, showing that building collision lists is the main performance bottleneck. Note that nvprof does not record time spent in OpenGL.

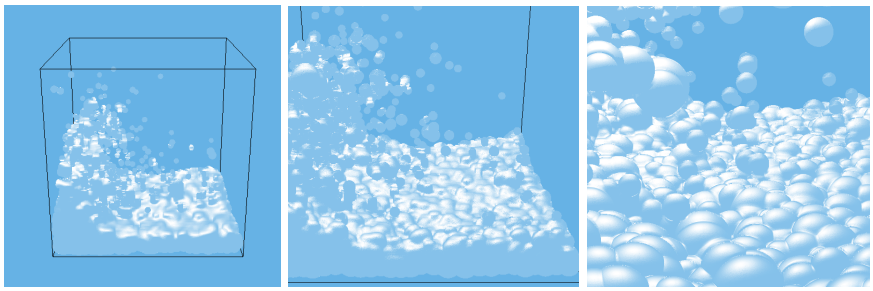

Figure 10: As the camera approaches the fluid, the fluid becomes less smooth since the particles become large relative to the bilateral filter smoothing radius.

5 Conclusion

We have implemented an SPH simulation and rendering pipeline that improves on existing work. While we are satisfied with the result, there is still room for optimization, such as by using occlusion culling to speed up rendering[7]. We might also be able to speed up the bottleneck kernel by using a lower-precision representation for particle positions.

Our project could also be extended to include all sorts of additional features, such as coupling with rigid body simulations or more complex boundaries. Following Green, the fluid could be made more visually appealing by adding translucency, shadows, caustics, reflection and refraction. A hierarchical approach could also be used to prevent the fluid surface from becoming less smooth when the camera gets too close to the fluid (Figure 10).

[7]Kubisch, C. (n.d.). Gl occlusion culling. Retrieved December 12, 2017, from `https://github.com/nvpro-samples/gl_occlusion_culling`

Maximum Euler time step	1.0
Gravity ($\|\vec{g}\|$)	0.0005
Particle radius (R)	2.0
Steady-state density (ρ_0)	1.1
Tait incompressibility factor (γ_{Tait})	2.0
Contact force coefficient ($F_0^{(c)}$)	0.01
Viscosity coefficient ($F_0^{(v)}$)	0.03
Boundary viscosity coefficient	0.3
Boundary pressure (P_b)	1.2
Coordinate precision (p)	8
Maximum collision list size	32
Schlick reflection coefficient (R_0)	0.133
Shininess (γ)	3.0
Light scaling coefficient (L_0)	6.0
Brightness offset	0.2
Bilateral filter kernel size (M)	7
Bilateral filter spatial smoothing factor (σ)	7.0
Vertical depth smoothing factor (σ'_y)	0.00004
Horizontal depth smoothing factor (σ'_x)	0.00001
Near plane (z_{min})	0.1
Far plane (z_{max})	1000
Bounding box width	50.0

Figure 11: Parameters used in our implementation.