Introduction

In this project, we implement a Smoothed Particle Hydrodynamics (SPH) method to simulate water. Our implementation follows closely the method described in [MCG03]. The code also includes other performance optimizations, such as Z-indexing and sorting [IABT11], SIMD vectorization, and multicore parallelization. We test the implementation with six test scenes simulating different situations involving water, and show that our system is efficient and scalable.

Related Work

SPH is used in computer graphics to simulate fluid interactively for the first time in [MCG03]. In this work, the authors try to achieve the incompressible property of water by modeling pressure using a gas equation. This spring-like model needs a high stiffness value to ensure incompressibility, but that implies stricter time steps to deal with very large pressure forces. In practice, a moderately small time step is often used, making the water compressible. Follow-up works focus on improving the method's performance, and ensuring incompressibility while using similar or larger time steps. They include [BT07] (Tait's equation for pressure computation to make the water less compressible), [APKG07, ZSP08] (adaptive particle sizes to avoid simulating too many particles in unimportant regions), [SP09, CBP05] (predictive-corrective pressure (PCISPH) and density relaxation, allowing much larger time steps), [IAGT10] (smarter boundary handling method for PCISPH), [AIY04, HKK07a, HKK07b] (GPU simulation), [GSSP10] (CUDA implementation, Z-indexing), [IABT11] (efficient implementation on multi-core CPUs). In the proposal for this project, we planned to implement PCISPH [SP09], but were unable to do so due to time constraint.

Smoothed Particle Hydrodynamics

To simulate fluid using particles, we begin with the Lagrangian version of the Navier-Stoke equations. The acceleration for each particle is calculated using the following equation.

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2\mathbf{v} + \frac{\mathbf{f}_{ext}}{m} + \mathbf{g}$$

The second equation which states the incompressibility condition is difficult to enforce using a particle system, thus is hardly used. In practice, incompressibility is usually handled by empirical techniques. To convert the above equation to SPH style, we start with the discrete definition of SPH, which is a method to interpolate field quantities using some kernel function W.

$$\langle A \rangle_i = \sum_j \frac{m_j}{\rho_j} A_j W(r_{ij})$$

The derivatives of the quantity are evaluated by putting the gradient or Laplace operator inside the kernel. We can now rewrite the first (pressure) and second (viscosity) terms on the right hand side of the Navier-Stoke equation in SPH style as follows.

$$\left\langle -\frac{1}{\rho}\nabla p \right\rangle = \sum_j P_{ij}\, \nabla W_p(r_{ij})$$

$$\left\langle \frac{\mu}{\rho}\nabla^2 \boldsymbol{v} \right\rangle = \sum_j \boldsymbol{V}_{ij} \nabla^2 W_v(r_{ij})$$

Each SPH method is free to design its own formula for P_{ij}, $\mathbf{V}_{ij}$, $W_p(r_{ij})$, and $W_v(r_{ij})$. Our implementation uses the set of equations described in $[\mathrm{MCG03}]$ to interpolate pressure and viscosity.

Implementation

The code for this project is written in C++. Other libraries are used for specific purposes. They are: OpenGL (for rendering), Freeglut (for OS and I/O functions), Parallel Pattern Library (for threading), and Eigen (for SIMD vectorization). SPH is implemented as a typical particle system where the particles interact with one another and each particle reacts to forces exerted by nearby particles. Our algorithm is summarized by the pseudo-code below. The main difficulties come from the facts that SPH relies on correct parameters to work, and that the performance decreases fast with increasing number of particles.

```
while animating do
  for all $i$ do
    find neighborhood $N_i(t)$
    compute density $\rho_i(t)$
    compute pressure $p_i(t)$
  for all $i$ do
    compute forces $\boldsymbol{F}_i^{p,v,g}(t)$
  for all $i$ do
    compute new velocity $\boldsymbol{v}_i(t+1)$
    compute new position $x_i(t+1)$
```

Parameters

SPH is sensitive to its parameters and initial condition. First of all, the Navier-Stoke equations are scale-sensitive. In practice, only a relatively small volume of water can be simulated interactively. Therefore, we use a scaling factor to relate the SPH simulation world to the rendering world. Such a parameter is needed because it makes integrating the water with other objects in the world easier. The scaling factor used in our implementation is taken from a presentation by AMD at http://developer.amd.com/zones/OpenCLZone/Events/assets/SmoothedParticleHydrodynamics.pdf.

Another important factor is the boundary handling method. Simply pushing the particles back when it penetrates the boundary walls may not work since doing so would change the density of water abruptly and hence affect the system's stability. Reflecting a particle's velocity along the contact normal is also not enough to deal with very high pressure forces in most cases. In our implementation, the boundaries exert a very stiff spring-like force ($\boldsymbol{F}$) to push the particles away along the contact normal ($\boldsymbol{n}$). Also, a dampening factor is included to model energy loss when collision happens. The drawback of this approach is that since $\boldsymbol{F}$ is often very large, a small time step is required. This is not a problem in our case, however, since the use of gas equation for computing pressure already requires small time steps.

SPH is also sensitive to its initial condition. For the system to work correctly, the initial positions of particles need to be carefully calculated. Placing them too far apart will result in some particles having no neighbors and thus moving unpredictably; placing them too close together can make the system explode due to abnormally high pressure. The initial gap between the particles (and also the mass of each particle) can be calculated using the fluid's rest density, the total number of particles, and the volume the particles occupy initially.

A particle's acceleration is caused by viscosity, pressure, gravity, and maybe external forces. In some scenarios, the pressure force can sometimes get too high due to abnormally low/high density. To avoid explosion, we artificially dampen the acceleration of each particle if it gets larger than some predefined value. The gravitational acceleration is only added after this dampening to avoid it getting "lost" in some close-to-infinity value. We finally find the particles' position in the next simulation step using leap-frog integration, which computes velocities and positions at interleaving time steps. The advantage of this integration scheme is that it is accurate to the second order and is relatively cheap to compute, although it requires more memory to keep track of a particle's velocity and position in previous time step.

All the above complications, together with the fact that the parameters need to be all working together, make parameter tuning a significant part of the project. In our implementation, the user needs to specify the following parameters for the system: the total number of particles, the rest density of water, the pressure's stiffness, the viscosity coefficient, the kernel radius, the time step, the simulation scale, the wall's stiffness, the wall's dampening coefficient, and the maximum acceleration of a particle. Some parameters are explained above; for the rest of them, we refer the reader to [MCG03]. We borrow default values for some of the parameters from various sources on the Internet (most of them are mentioned above); the rest of the values are found by experiments. The details are also documented in the code.

Data Structure

In SPH, each particle needs to query for its neighboring particles a few times during a simulation step. This operation is the main bottleneck of the method; a naïve implementation would need to iterate through all the particles for each particle, making the runtime complexity $O(n^2)$. A common technique to deal with this problem is to use a grid data structure to manage the particles. The whole simulation space is divided into uniform cells, each cell stores the particles inside it. For the system to scale, care must be taken when designing the data structure for the grid.

In an early version of our system, the grid is implemented using C#'s `Dictionary`, which is a hash table. The key used to index into the hash table is a tuple of cell coordinate <x, y, z>. The value stored with each key <x, y, z> is the list of particles belong to the cell <x, y, z>. In theory, this implementation has an advantage that is less memory is used for the grid, since only few cells are occupied by particles. In practice, however, the cost of the indexing (look up) operation is high, especially when we need to look up a large number of particles per time step. That is because it takes time to compute a hash value for each key, and also because when hash collision happens, the algorithm needs to traverse a linear list of items to find the right value.

The next data structure we try for the grid is a C++ array where each element stores a linked list (`std::list`) of references to particles. However, a linked list is not the fastest data structure to use if we want to iterate through the elements, which is what we do a lot in SPH. The reason is that the elements in a linked list may not be consecutive in memory, thus looping over all elements will cause many cache-misses. Writing cache-friendly code is also the reason why we use C++ instead of C#, as the latter manages objects by pointers exclusively, giving the programmer little control over the memory location of the actual objects.

The data structure that we settle on is an array where each element stores a vector (using `std::vector`) of references to particles. In contrast to a linked list, elements in a vector are stored consecutively in memory, thus cache-misses when iterating through the elements are kept to a minimum. One complication when using vectors, however, is that the delete operation is not constant but linear time complexity. We avoid using this operation by keeping track of the number of elements manually instead of relying on the `std::vector` class to do so for us.

Z-Indexing and Sorting

Z-indexing (or morton encoding) is a method to map multi-dimensional values to one-dimensional value, while preserving locality. Consider the following example. Suppose our grid is 8x8 in 2D, and is stored in memory as a 64-element one-dimensional array. Using a trivial indexing scheme, a particle at <4, 3> would be stored in grid[4 * 8 + 3] or grid[35], while a particle at <5, 3> would be stored in grid[5 * 8 + 3] or grid[43]. The two particles are close to each other in 2D but are stored far apart in memory. In SPH, we often need to loop through sets of neighboring particles. As we saw in the previous section, storing nearby particles in far-apart memory locations is bad for this looping operation since it is not cache-friendly. Z-indexing, on the other hand, uses bit-interleaving of coordinates to compute a 1D index. In our example, the bits of 4 (100) and 3 (011) are interleaved to produce 26 (011010), while 5 (101) and 3 (011) will give us 27 (011011). Thus the two particles would be stored next to each other in memory, which means iterating through the particles results in less cache misses.

Using z-indexing can improve the grid data structure's performance. However, the grid only stores the handles (references) to particles. To fully utilize the cache, the particles themselves need to be stored in a similar way as the handles. We follow [IABT11] and sort the particle array once every 100 simulation steps using insertion sort. The array need not be sorted very frequently since particles' positions change slowly over time, while insertion sort is used since it is very efficient for almost-sorted arrays.

SIMD Vectorization

A lot of operations on the particles are vector operation since the velocities, positions, accelerations, and forces are all vector quantities. Modern processors can perform vector operations very efficiently using special instruction sets. One example of such an instruction set is the Intel's Streaming SIMD Extensions (SSE). As an example, consider the addition of two 3D vectors, a and b. Using normal instructions, we write this as `c.x = a.x + b.x; c.y = a.y + b.y; c.z = a.z + b.z;` which will translate into 3 or more machine instructions. Using SIMD vectorization, all three instructions can be reduced to one, provided the vectors (`a`, `b`, and `c`) are stored in a special way in memory so that the processor can fetch and add them component-wise. Obviously this is a major performance improvement over the traditional code. In our implementation, we use a library called Eigen (http://eigen.tuxfamily.org). This linear algebra library compiles to SSE2 instructions, and also simplifies the code dealing with vectors significantly (adding two vectors is simply written as `c = a + b;`).

Multithreaded Programming

SPH is an embarrassingly parallel problem, since the particles do not write to shared variables. As such, we can easily, for example, assign each particle to a dedicating thread so that the whole system can take advantage of multicore processors. We rely on Microsoft's Parallel Pattern Library for C++ to do the threading for us, simply by rewriting all each `for` loop over all particles into a `parallel_for` loop. The PPL library will decide how many threads to spawn and how to map particles to threads effectively.

Result

We test our SPH simulation with six test scenes. They are described below.

- Dam break. A column of water initially occupying half of tank is collapsing, creating back and forth waves from one side of the tank to the other.
- Water drop. A volume of water is initially put in the air, above a water bed. It then falls down, creating some waves.
- Sink. A volume of water is falling down to the level below through a hole in the center.
- Wave. A tank initially contains some water. One of the walls move back and forth periodically, creating large waves to the other side.
- Ball. A ball is dropping on a volume of water. The ball then move back and forth periodically, pushing the water around.
- A column of water is put into a container with the U shape (⊔). The water flows from one side to the other until equilibrium is reached.

In all scenes we use 6400 particles inside a volume of roughly 0.008 m^3. However, not all scenes use the same set of values for the parameters. All the scenes are captured in a video submitted together with this report.

For performance evaluation, we run the dam breaking scene on an Intel Core2 Duo 2.0GHz system with 2GB DDR2 RAM and a GeForce 8600M GT graphics card. With 6400 particles the frame rate is 22.15 fps, with 24000 particles, the demo runs at 4.85 fps. These results show that our system's performance is linear with respect to the number of input particles, which means our use of data structure is efficient. Due to time constraint, however, we are unable to evaluate scalability of the system on a machine with more than two cores, and to quantify the effects z-ordering and SIMD vectorization on performance.

References

[APKG07] Adams B., Pauly M., Keiser R., Guibas L. J.: Adaptively Sampled Particle Fluids. In *ACM Transactions on Graphics* 26, 3 (2007), 48—54.

[AIY04] Amada T., Imura M., Yasumoro Y., Manabe Y., Chihara K.: Particle-Based Fluid Simulation on GPU. In *ACM Workshop on General-Purpose Computing on Graphics Processors* (2004).

[BT07] Becker M., Teschner M.: Weakly compressible SPH for free surface flows. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 63—72.

[CBP05] Clavet S., Beaudoin P., Poulin P.: Particle-Based Viscoelastic Fluid Simulation. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 219—228.

[GSSP10] Goswami P., Schlegel P., Solenthaler B., Pajarola R.: Interactive SPH Simulation and Rendering on the GPU. In *SCA '10: Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

[HKKA07a] Harada T., Koshizuka S., Kawaguchi Y.: Smoothed Particle Hydrodynamics on GPUs. In *Proceedings of Computer Graphics International* (2007), 63—70.

[HKK07b] Harada T., Koshizuka S., Kawaguchi Y.: Sliced Data Structure for Particle-Based Simulation on GPUs. In *GRAPHITE '07: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, 55—62.

[IABT11] Ihmsen M., Akinci N., Becker M., Teschner, M.: A Parallel SPH Implementation on Multi-Core CPUs. In *Computer Graphics Forum* 30, 1 (2011), 99—112.

[IAGT10] Ihmsen M., Akinci N., Gissler M., Teschner, M.: Boundary Handling and Adaptive Time-Stepping for PCISPH. In *Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS 2010), 79—88*.

[MCG03] Muller M., Charypar D., Gross M.: Particle-Based Fluid Simulation for Interactive Applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 154—159.

[SP09] Solenthaler B., Pajarola R.: Predictive-Corrective Incompressible SPH. In *ACM Transactions on Graphics* 28, 3 (2009), 40:1—6.

[ZSP08] Zhang Y., Solenthaler B., Pajarola R.: Adaptive Sampling and Rendering of Fluids on the GPU. In *Proceedings of the 2008 IEEE/Eurographics Symposium on Point-Based Graphics*, 137—146.