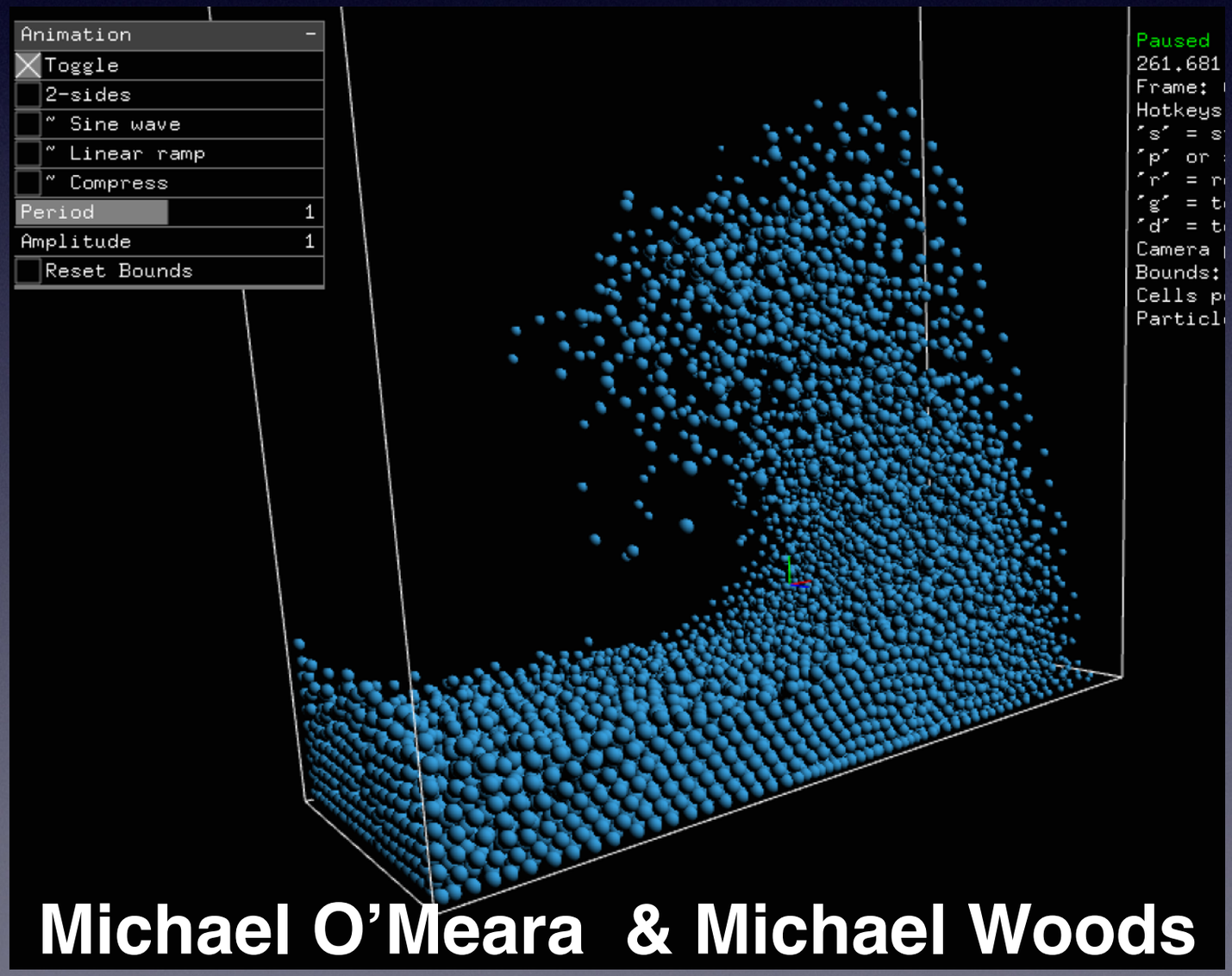
Real*(ish)*-time fluid simulation with Position Based Fluids



Michael O'Meara & Michael Woods

Motivation

- Fluids are visually impressive…
- …but computationally expensive to simulate
- The semi-Lagrangian method described by Müller/Bridson's SIGGRAPH 2007 notes works well, but relies on a grid, making fluid interactions with objects difficult

Enter: Position Based Fluids

- Introduced by Müller and Macklin (2013)
- Particle-based system
- Based off the Position Based Dynamics framework developed by Müller, Heidelberger, Hennix, and Ratcliff (2006)
- **Pro**: unconditionally stable with large timesteps—a necessity for realtime simulation
- **Con**: not physically accurate, but still looks good (which is what we care about anyway)

Technical Considerations

- Must be fast (> 15 FPS) for ~10K particles
- How can we do this?
- Take note: calculations per particle are generally independent of one another
- One of those “embarrassingly parallel” kind of problems*

***mostly true**

GPU acceleration & OpenCL

- All simulation steps implemented as OpenCL *kernels* executed in parallel by GPU threads

- Each kernel roughly corresponds to a

```
for all particles i do
  …
end for
```

  block in the main PBF simulation loop

- Several thousand threads can be executed simultaneously in hardware

- Outcome: 1 thread = 1 particle

- Downside: OpenCL is a restricted subset of C99—that means no dynamic memory allocation, no STL, nothing fancy—just structs, arrays, and pointers

Algorithm 1 Simulation Loop

1: **for all** particles i **do**
2: apply forces $\mathbf{v}_i \Leftarrow \mathbf{v}_i + \Delta t \mathbf{f}_{ext}(\mathbf{x}_i)$
3: predict position $\mathbf{x}_i^* \Leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
4: **end for**
5: **for all** particles i **do**
6: find neighboring particles $N_i(\mathbf{x}_i^*)$
7: **end for**
8: **while** $iter < solverIterations$ **do**
9: **for all** particles i **do**
10: calculate λ_i
11: **end for**
12: **for all** particles i **do**
13: calculate $\Delta \mathbf{p}_i$
14: perform collision detection and response
15: **end for**
16: **for all** particles i **do**
17: update position $\mathbf{x}_i^* \Leftarrow \mathbf{x}_i^* + \Delta \mathbf{p}_i$
18: **end for**
19: **end while**
20: **for all** particles i **do**
21: update velocity $\mathbf{v}_i \Leftarrow \frac{1}{\Delta t}(\mathbf{x}_i^* - \mathbf{x}_i)$
22: apply vorticity confinement and XSPH viscosity
23: update position $\mathbf{x}_i \Leftarrow \mathbf{x}_i^*$
24: **end for**

Key: fast neighbor search

Uniform Grid using Sorting

NVIDIA

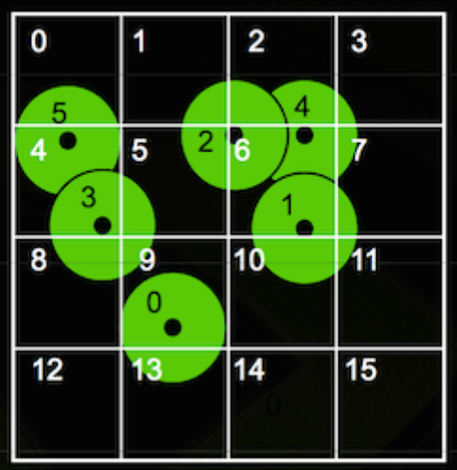
- Grid is built from scratch each frame
  - Future work: incremental updates?
- Algorithm:
  - Compute which grid cell each particle falls in (based on center)
  - Calculate cell index
  - Sort particles based on cell index
  - Find start of each bucket in sorted list (store in array)
  - Process collisions by looking at 3x3x3 = 27 neighbouring grid cells of each particle
- Advantages
  - supports unlimited number of particles per grid cell
  - Sorting improves memory coherence during collisions

Slide courtesy of Simon Green/NVIDIA, 2008
http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_ParticleFluids.pdf

Fast neighbor search: an example

Example: Grid using Sorting

NVIDIA



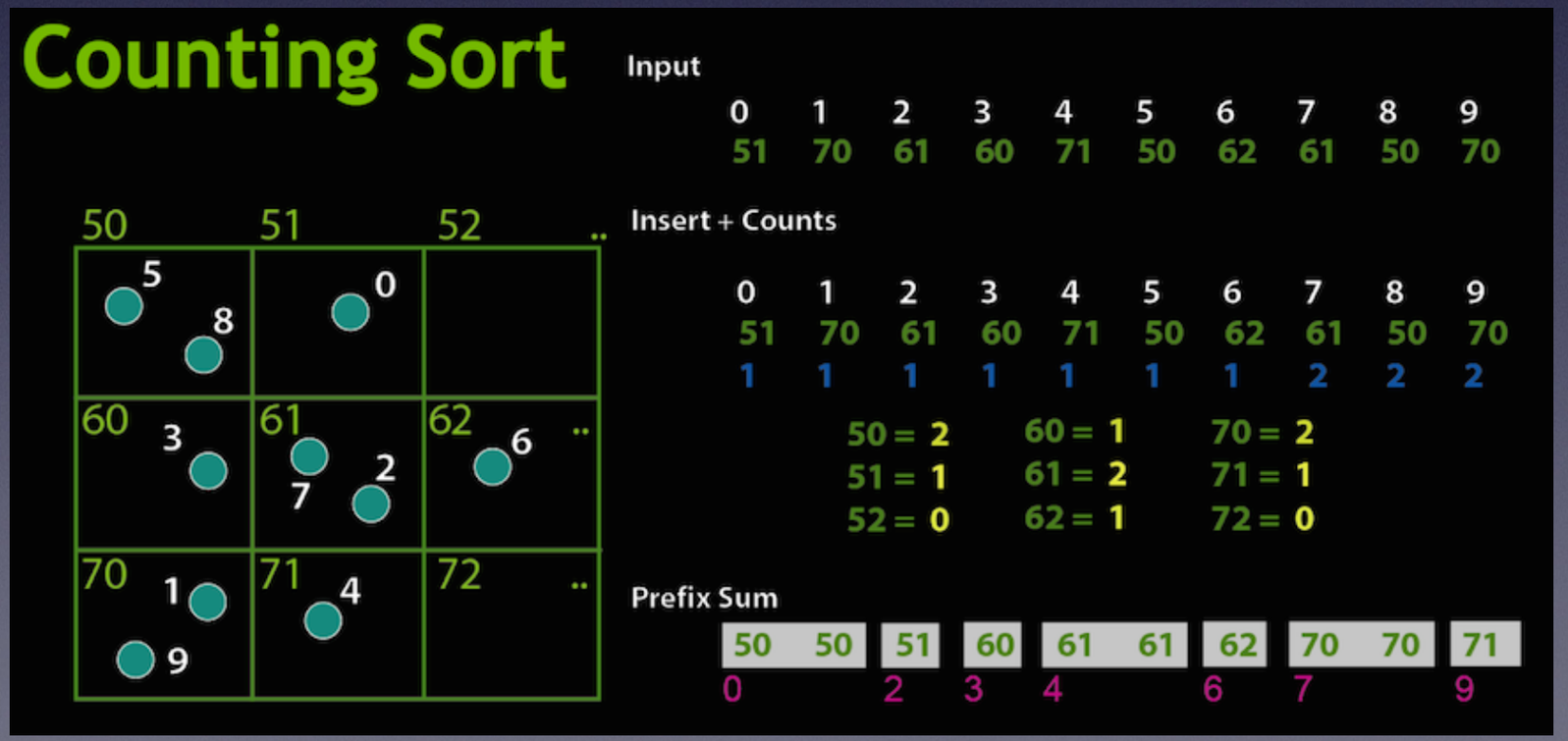
unsorted list (cell id, particle id)	sorted by cell id	cell start
0: (9, 0)	0: (4, 3)	0: -
1: (6, 1)	1: (4, 5)	1: -
2: (6, 2)	2: (6, 1)	2: -
3: (4, 3)	3: (6, 2)	3: -
4: (6, 4)	4: (6, 4)	4: 0
5: (4, 5)	5: (9, 0)	5: -
		6: 2
		7: -
		8: -
		9: 5
		10: -
		...
		15: -

Slide courtesy of Simon Green/NVIDIA, 2008
http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_ParticleFluids.pdf

Our implementation

- Macklin/Müller use the method outlined by Green 2008, which uses radix sort for sorting particles by cell

- We used a method outlined by Hoetzlein, 2013 using counting sort. Otherwise, our approach is similar to Green's



Slide excerpt courtesy of Rama Hoetzlein/NVIDIA, 2013 "Fast Fixed-Radius Nearest Neighbor Search on the GPU"
http://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf

Our implementation, cont'd

- **Pro**: Counting sort is easy to implement (if you have a parallel prefix-sum implementation on hand) and fast:
  (1) Compute the particle-to-cell histogram array
  (2) Compute the prefix sum array
  (3) Reorder particles based on prefix sum array

- **Con**: hard if you *don't* have a parallel prefix-sum implementation on hand. Not many off-the-shelf implementations exist. Luckily, Apple has one available as part of a tutorial on OpenCL*

*https://developer.apple.com/library/mac/samplecode/OpenCL_Parallel_Prefix_Sum_Example/Introduction/Intro.html

More technical details

- Rendering particles as spheres is slow and unnecessary
- There's a cost associated with sending geometry and transformation matrices to the GPU
- Instancing is one approach, but a transformation matrix is still needed per particle position
- Key idea: particles just need to *look* like spheres (or anything, for that matter)
- Solution: render a set of vertices using `GL_POINTS` and create the illusion of spheres using point sprites at the shader level
- Thanks to Harmony for the idea

Shortcomings and Improvements

- OpenCL driver/SDK implementation quality is spotty across vendors. Same code should run everywhere, but doesn't always
- Simulated surface tension induced by artificial pressure in our implementation is lacking in realism. This might be due to the size of the particle neighborhood being examined
- Simulation could be faster. Ideally, we would like to run it in real-time with ~30K particles…currently, not fast enough for real-time usage
- Wanted to experiment more with different viscosity/vorticity coefficient tweaking in real-time

Check it out!

The source code and all supplemental information is available on Github

https://github.com/mikeswoods/cis563-final-project

Thanks!