Team Grus

github.com/timsl/agp-grus

Victor Ähdel
Tim Olsson
Konrad Magnusson
Alex Sundström

Creation of the Moon

Simulate planet collision

N-body

- N bodies that interact with each other
  - according to force table
- All-pairs vs. Hierarchical methods (with far-field approximations)
  - We stick with all-pairs as in the original paper (for accuracy)
  - $O(n^2)$ vs. $O(n \log n)$

Implementation

- OpenGL
- CUDA
- GLM - matrix math
- GLFW - window drawing
- GLAD - capability detection

Struct oriented / Component-based

- We don’t like UML/inheritance
- WorldState
  - CameraState
  - WindowState - GLFW
  - HeldActions - Held keys, to move
  - GPUState - CUDA
  - Sphere (one) - OpenGL

OpenGL Visualization

- Based spheres on the OpenGL lab codebase (GLFW version)
- Sphere struct
  - Handles rendering
  - Creates vertices and indices only once
  - Has its own VAO
- Face culling

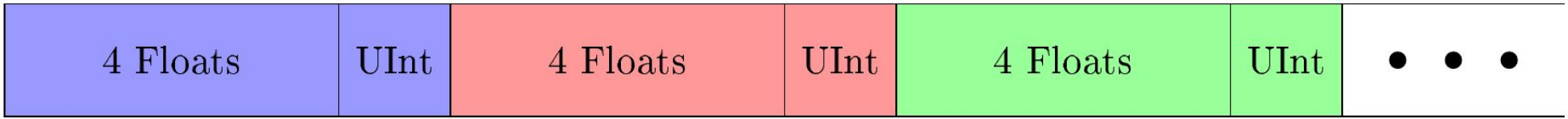
Instanced Rendering

- We want to render many particles very fast
- Reduce CPU overhead
- Need to store all particle positions on the GPU

Instanced Rendering - VBO

- We can create a large VBO which holds position and type values
  - Type is for which color we render the sphere as
- We can pack these values using **glVertexAttribPointer** and assigning a stride and offset
  - Need **glVertexAttribIPointer** for the type since it's an integer
- **glVertexAttribDivisor** set to 1, we advance one "stride" for every instance, not every vertex
- The VBO is nr_particles * (4 * GL_FLOAT + GL_UINT) large
  - Used a char instead of uint first, but was problematic to use with interop.

Instanced Rendering - VBO



Instanced Rendering - Rendering

- With **glVertexAttribPointer** we also defined a attribute index
- Enable that index with **glEnableVertexAttribArray**
- We call **glDrawElementsInstanced** to draw the particles
  - How many CPU calls for every frame depends on how many “stacks” the sphere is made out of
  - Code inherited from the OpenGL lab
- Compute view-projection matrix on CPU

Instanced Rendering - Vertex Shader

- In the shader the aforementioned attribute index helps define our in layout
  - layout(location = 0) in vec3 pos;
  - layout(location = 1) in vec4 M;
  - layout(location = 2) in uint type;
- Take the view-projection matrix as a uniform
- Send the type as an out parameter with the “flat” prefix
  - No interpolation

Instanced Rendering - Fragment Shader

- Has a uniform color array of vec4
- Use the type uint to select the color

Instanced Rendering - Our method vs. canvas

- On canvas: Use shader storage buffer object (SSBO)
  - Can define structs
- Our method works on OpenGL 3.3+
- SSBO 4.5+
- However SSBO looks a lot easier to maintain for the programmer
  - Structs instead of pointer math
  - Less confusion = good

CUDA

- We use it.
- GPUs are speedful
- More or less follow the nvidia gems book
  - Shared memory slightly improved performance
  - **body_body_interaction()**
  - **calculate_forces()** - loop body_body
  - **apply_forces()** - leapfrog integration
- By far the most computationally intensive part

CUDA Optimizations

- Found bottlenecks with nvprof/Nvidia Visual Profiler
  - **pow()** is slow
- Branch elimination
- Floats
  - Thought we needed double precision for some force parts at first
- Optimal blocksize

Interop

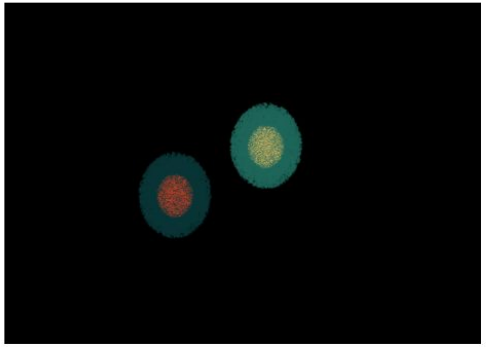
- CUDA - OpenGL
- A lot less memory transfer & synchronization

Particle changes

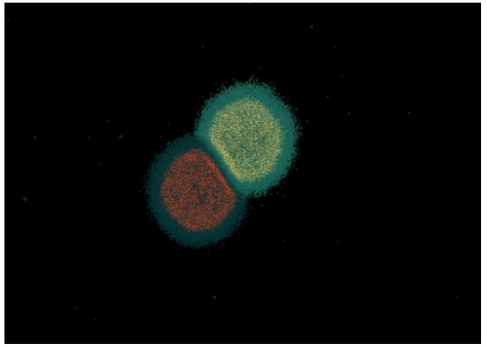
- 131k particles < 30fps
- Less particles
  - Change D
  - Change gravity/collision ratios

Simulation Results

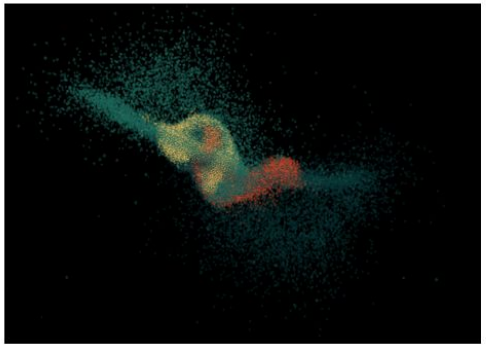
- 32,768 particles
- Sensitive to parameters



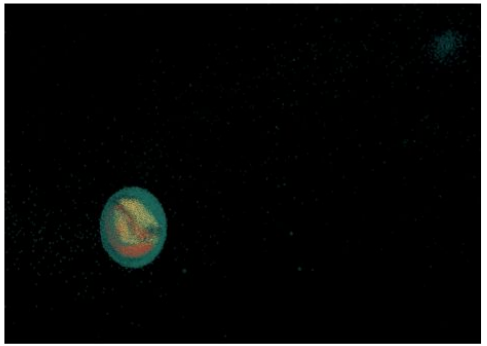
(a) Initialization



(b) First impact



(c) After collision



(d) Planet and something moonlike

Video demo

- 65536 (2^16) particles
- Bad parameters
- NVIDIA GTX 1070 (1920 CUDA cores, ~ 6.4 TFLOPS)
- https://www.youtube.com/watch?v=aJSygonprBI

Live demo

- 10240 (10·2^10) particles
- Exquisite parameters
- NVIDIA 840m (384 CUDA cores, ~ 863 GFLOPS)
- Spaceship camera

CUDA - Performance results

Kernel	Time	%
calculate_forces	57541	99.85%
apply_forces	52	0.09%
update_gl	31	0.06%

- Mean update time (μs) (CUDA) vs blocksize
- NVIDIA GTX 960 4 GiB (1024 CUDA cores, ~ 1.5 TFLOPS)
- Approx O(n^2)

Particles	8	16	32	64	128	256	512	1024
1024	574	478	454	443	439	438	447	508
2048	1552	926	788	760	744	737	743	843
4096	5124	2631	1697	1582	1514	1483	1456	1626
8192	19724	9951	5084	4359	3968	3811	3711	3655
16384	77976	39116	19784	18670	17437	17000	14660	14350
32768	311170	155934	78569	77015	70697	68759	58424	57163
65536	1248198	626355	314103	281629	259492	252709	234937	231091

OpenGL - Performance results

- Mean update time (μs) (OpenGL)
- NVIDIA GTX 960 4 GiB (1024 CUDA cores, ~ 1.5 TFLOPS)
- Approx O(n)

Particles	Mean	Std.Dev.
1024	428	428
2048	555	413
4096	833	556
8192	1274	681
16384	2280	805
32768	4108	894
65536	8155	912

Discussion

- Feels pretty fast for $O(n^2)$
- Never found good parameters
- GPUs performance necessary
  - CUDA was pleasant, close to C++
- OpenGL is ready for >100k particles

Possible Improvements

- Don’t use all-pairs
- Search longer for good parameters
- Render text with performance information
  - Had in plan on first prototype, but no time / was not very necessary
  - May need separate shaders
- Multithreading

Questions?