N-Body Simulator

Controls:
Esc - Exit program

WSAD - Move camera around in horizontal plane
. or > - Move camera upwards
, or < - Move camera downwards
/ or ? - Hold for precise movement
Mouse - look around (Buggy)

Q - Toggle wireframe
B - Toggle blending
P - Toggle rendering as GL_POINTS

= or + - Increase simulation time step
- or _ - Decrease simulation time step
0 or) - Reset simulation time step

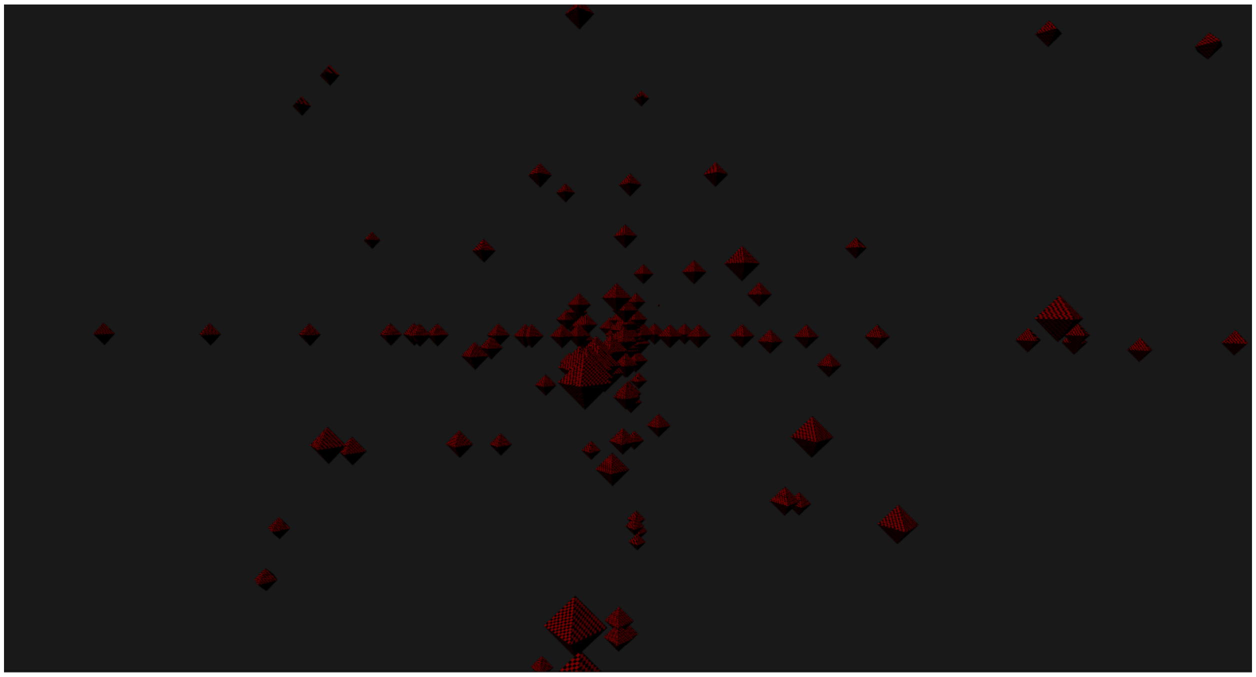
Setup:
OS: Linux 3.8.0-35-generic, Mint 15
Compiler: g++ (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3
GPU: Radeon HD 6950 with latest AMD drivers

Should work with any Linux environment if the GPU and drivers support OpenGL 4.3.
While it could compile on OSX with minimal change, I highly doubt Macs support OpenGL 4.3.
Could probably be ported to Windows, but I have no idea how.

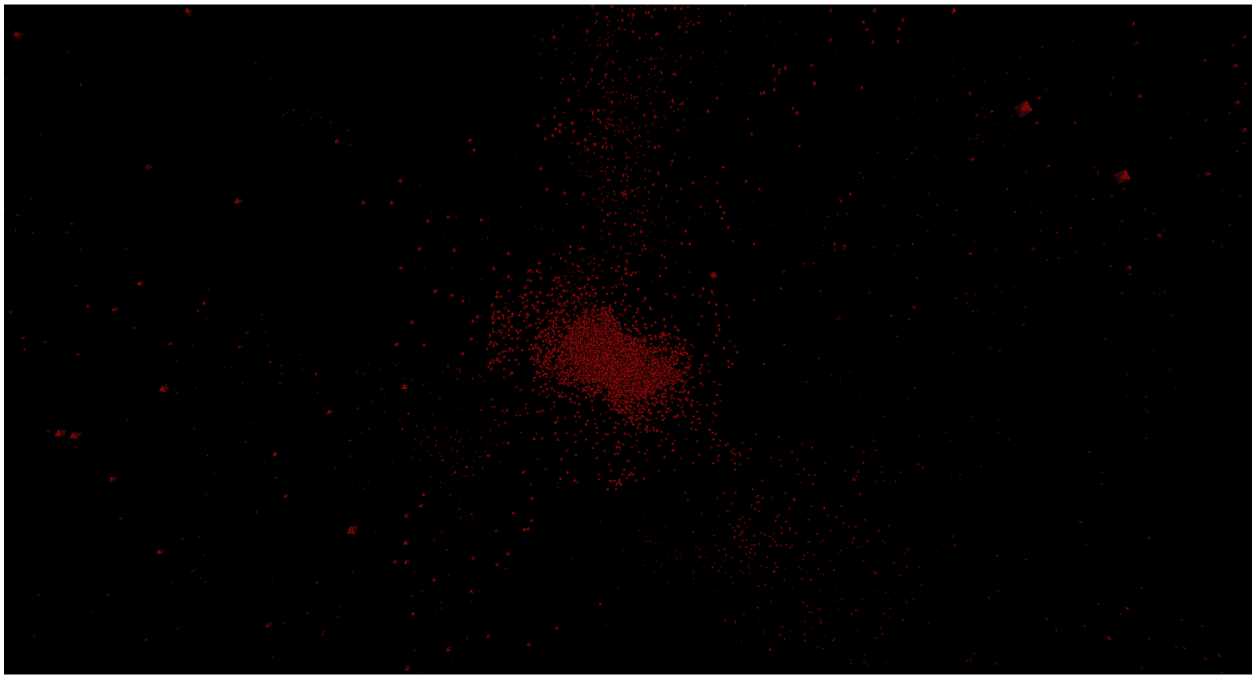
Video:
https://www.youtube.com/watch?v=ZnsQjs9ID8w
For best results, watch in HD.

Process:
The initial goal of the project was to meet the requirements of the specification, geometry, lighting and texturing. All geometry is loaded in from and obj file as this gives flexibility. Lighting encompases ambient, diffuse and specular highlights within the fragment shader. Texturing is kept simple and is read through each objects mtl file. Throughout the project an octahedron is used due to its simplicity which is good for render performance.

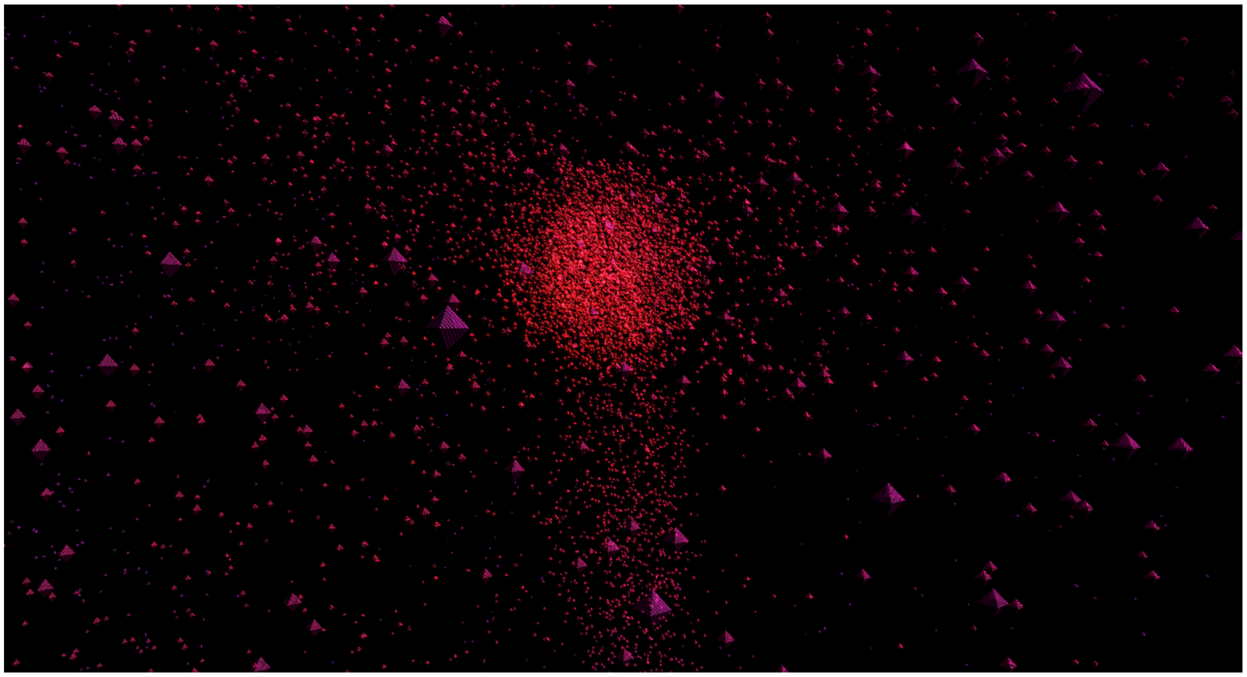
Once the basics were complete, it was time to implement the simulation aspects.
To achieve efficient drawing of multiple particles, instanced rendering was employed. This was chosen over other methods such as a geometry shader for it’s simplicity and equal, if not better performance. Using this method it was possible to smoothly render around 80000 particles simultaneously.
Now that there was a method to draw multiple particles, the actual simulation was implemented. Rather than stepping straight to a full blow compute shader, a simple implementation was used temporarily. This involved mapping the simulation data buffer each frame to update the position and velocity of each particle. While this did work, performance was atrocious giving a maximum of 800 particles at acceptable framerates. This is because large amounts of data had to be passed between the CPU and GPU each frame.
The screenshot below shows 300 particles. Notice the tendency for particles to sit on the axes. This was a bug relating the vertex attribute pointers.

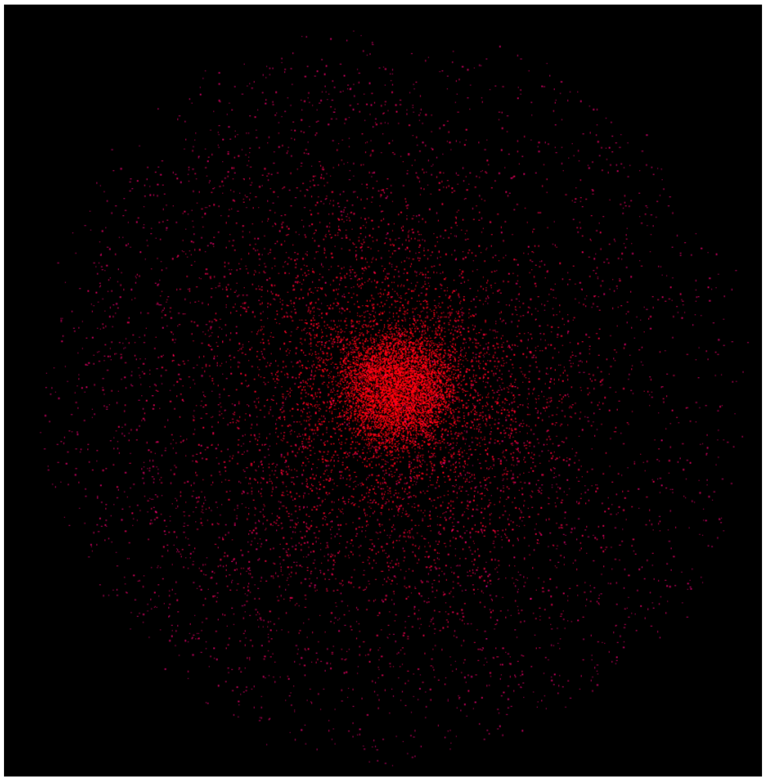


Since the basics were implemented it was time to implement the first major feature, a compute shader. This addition included significant restructuring of the code by in introducing a particle class. As a performance consideration, the particle system was designed from the outset with a double buffer for the position and velocity data. This was chosen as it opened the opportunity for simultaneous computing and rendering.
Initially texture buffers were used to contain the simulation data. However, these were later changed to shader storage buffer objects due to their simplicity. This decision introduced a dependency on OpenGL 4.3 which in turn hampers code portability. However this was allowed since compute shaders already require OpenGL 4.3.
The screenshot below shows around 20000 particles being simultaneous rendered and computed. Note that the basic n^2 algorithm for simulation imposes a severe bottleneck on performance, hence the original 80000 particles is impossible.



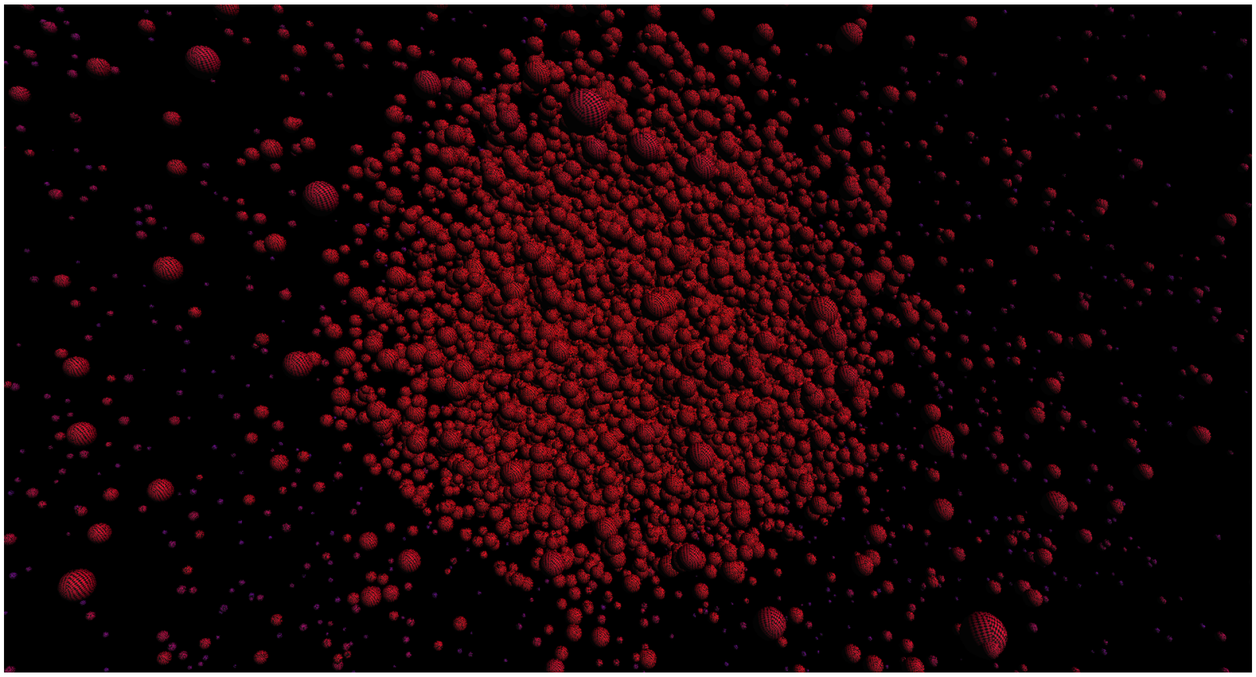
The next stage involved adding some cosmetic enhancements to the program. A simple addition in the fragment shader colors particles based on their distance to the origin. The alpha channel was then utilised and a form of alpha blending was used. The blending effect causes each particle to be see through and causes dense areas appear to glow white hot. Since each particle is transparent, the depth buffer is turned off as a performance improvement.
The other major addition in this stage was a dynamic camera which controls both the view and perspective matrices. The camera has its own velocity to give fluid movement. The camera may also be controlled through the mouse to look around, although this suffers gimbal lock.





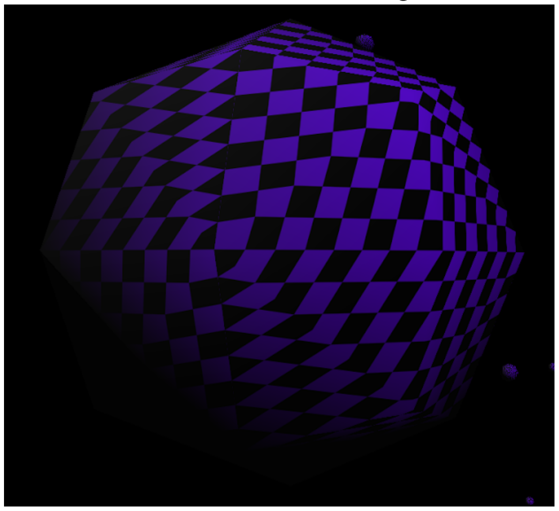
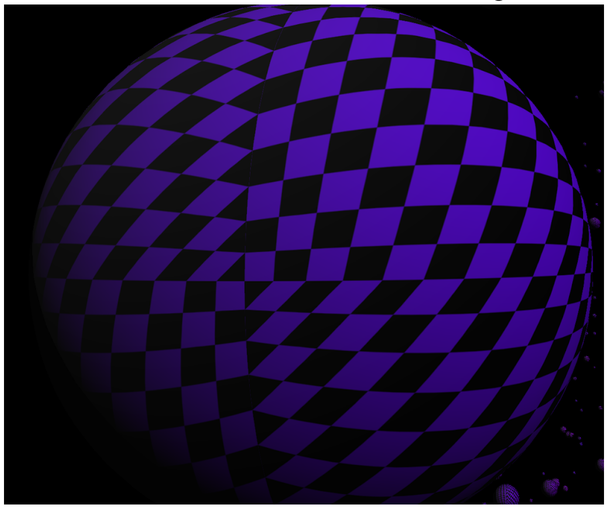
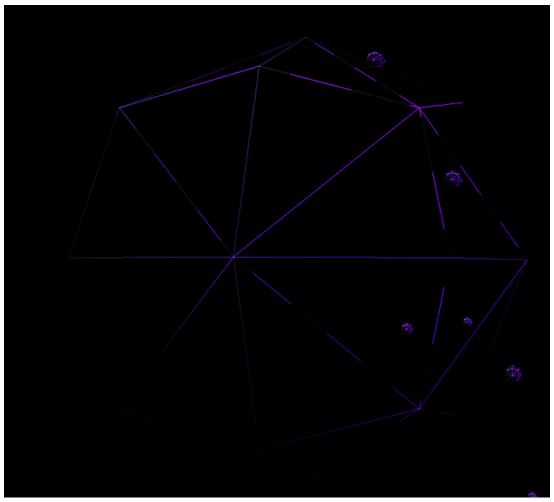
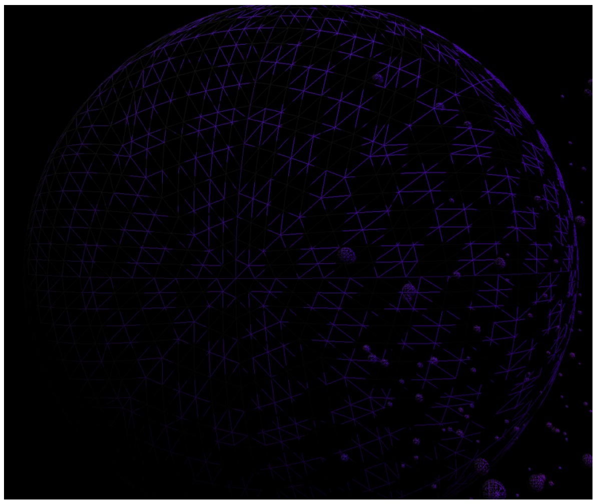
As more complicated graphics features were added, the rendering started to take away precious time from the simulator. This combined with the fact that there is no easy method to run the simulator independently of the frame rate was starting to degrade performance. To combat this issue a separate rendering program was implemented to simply render the particles as GL_POINTS with no complicated lighting or geometry calculations. These two rendering programs may be switched dynamically at run time for improved performance.

The last major feature added was the use of tessellation shaders to generate spherical particles. It was decided to use dynamic tessellation over multiple pre generated models of different complexity as sorting particles based on distance poses a problem. One method considered involved multiple rendering passes and discarding unwanted models with a geometry shader. This method introduces a lot of redundancy and the multiple draw calls which could potentially harm performance. The second option involved initially sorting each position into separate transform feedback buffer for each level of detail, then rendering each buffer separately. This option was later disregarded as overly complicated.
A tessellation shader overcomes this issue by generating each sphere dynamically from the original octahedron. One issue with this method is that particles generated at similar distances from the camera appear the same and hence there is redundant tessellation being performed.



The inclusion of tessellation puts some strain on the system due to the large amounts of interpolation required for the smooth shading of each sphere. To address this issue, the tessellation levels of each sphere is determined each frame based on the particles distance to the camera. This inclusion means that far away particles are only tessellated once while close particles may be tessellated up to 50 times to appear as a perfect sphere. This takes a lot of strain off the system since far away particles just look like dots and don’t require significant tessellation to appear as spheres.

The final feature of this project was implementing view frustum culling. The calculation of the normals is fairly straight forward and invalid particles have their tessellation levels set to 0, hence not being displayed.

1 level of tessellation, 48 triangles	~30 levels of tessellation, lots of triangles
	
	

Thanks for reading.
By Michael Hermann.