4.
My Program has no notion of air resistance or drag, acceleration will continue to increase velocity to infinity.

I have decided to implement simple laws of motion, and particularly gravity.

It is trivial to simulate the constant acceleration of gravity. I decided to do this on a unit scale where one tick of my program ($1/60^{th}$ of a second at full performance) is equal to one unit time. A suitable gravity acceleration $\boldsymbol{g}$ is selected to have the particles fall at a rate not too fast or not too slow for any observer watching.

A Particle updates its position every tick by adding its position and velocity vectors. Every tick, acceleration is also added to the velocity vectors to determine its velocity in the next tick.

There is also variance to the gravity applied on each tick, which is controlled by a program constant. This variance helps distribute the velocities of the particles slightly better, and make them seem to fall as though as they had variations in mass.

I believe the simple physics of my program is more than enough to simulate falling motion, and makes for a cool graphics generator. Higher fidelity simulation would only be a determent to the overall quality of the program, as most of the particles are only on the screen for at most a second, not enough to observe any intricate particle interaction. Furthermore, simulating particle interaction at the upper vertex bound of my program, 100,000+ particles, would be exponentially impractical with my current skillset, or even with WebGL.

5.
I believe my application does close to the minimum amount of work required for a CPU implementation of particle updates. Since most simulated particles are ones which are visible in the viewport (Those that aren't run out of time to live), I need to calculate the effects of velocity on every single particle and update their position. I have a for loop over the collective list of particles, which would be the minimum required to simulate particle motion.

However, in a GPU implementation, multiple particles will be able to be updated at once, drastically lowering the time taken for the updateParticles() call to be run. This would be ideal in a more complex system, especially one in which large numbers of particles interact with one another, eg. In a Supermassive Black Hole Star formation simulation.

I think CPU heavily bottleneck after about 66k verticies, as my GPU utilization at that stage was only 27%, however my CPU was at 40%+ which is usually the uperbound CPU usage of google chrome.
A game usually renders far more than 66k verticies in a scene at 60+ fps, therefore I think the bottleneck is in the CPU. I could make this effect less drastic by writing code with less cpu intensive instructions (for loops, repeated memory access), and more gpu intensive instructions (use buffers, manipulate the particles in buffers so they can be done in batch, rather than iterating over them with the cpu outside of the buffer, then loading it back into the buffer)

Looking at the advantage that rendering triangles has, even though it requires more GPU work, suggests that the bottleneck is indeed in the CPU, as a greater number of verticies can be processed per second as a result of moving some of the work to the GPU. Overall though I am pleasantly surprised at what WebGL on a browser is able to do with my hardware configuration.

FPS - Vertex Count

129	60		129	60
1285	60		1274	60
6425	60		10260	60
12849	60		16661	60
25700	58		23050	60
38500	47		30799	59
42405	43		38443	59
44975	37		44900	45
50115	33		51324	43
56539	26		57739	37
60395	21		66704	22
64251	17		74400	16
66821	14		80867	16
69391	14		85970	14
73245	14		89820	13
77101	13		94960	12
80955	12		100115	12
87381	11		102659	11
96375	9		106530	9