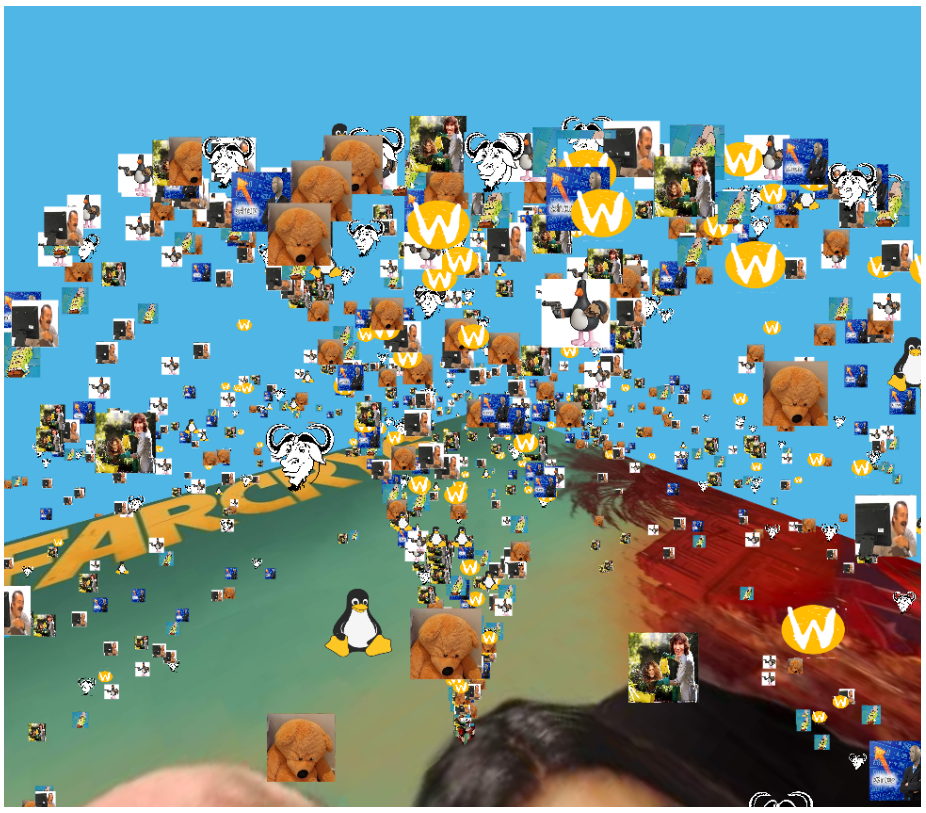
COSC 3P98 Assignment 3

Brett Terpstra - 6920201 - bt19ex@brocku.ca

April 6, 2023



Abstract

Particle systems are the cornerstone of all modern game engines as they allow effects such as smoke which are otherwise impossible with static meshes. Ergo a good particle system should be flexible while maintaining stable performance. This document serves as an informal report on the design and implementaion of two particle systems; a basic but flexible engine and a high performance extendable ”modern” GPU powered particle system.

Table Of Contents

1 Introduction **2**
1.1 Description 2
1.1.1 Extra Features 2
1.1.2 Missing Features 2
1.2 Building 2
1.2.1 Caveats 3
1.2.2 Build Commands 3
1.3 Usage 3

2 Performance Mode **4**
2.1 Design 4
2.2 Renderer 4
2.2.1 Rendering Pipeline 5
2.3 Compute Shader 5
2.4 Usage 6
2.4.1 Building 6
2.4.2 Running 6
2.5 Future Plans 6
2.5.1 Lists 6
2.5.2 Bitonic Sort 6
2.5.3 Tiling 7
2.5.4 Hierarchical Depth Buffers 7
2.6 Figures 8

Introduction

1.1 Description

Over the course of working on this assignment I began to wonder how far I could push my hardware and chapter 2 ”Performance Mode” will go furher into this. As for the main assignment requirements, they are met simply by running the CMake project. On my hardware the simple particle fountain can reach 30k particles and features sorted transparency.

1.1.1 Extra Features

- Ordered Alpha Blending
- ”Spray” Mode (12)
- Textured Particles/Plane/Cube (16)
- Particles with Different Textures (18)
- Extra Feature - ”Performance Mode” (23)

1.1.2 Missing Features

Random spin mode was left out intentionally for two reasons. One, I specifically designed the particle strucutre to fit in 32 bytes, half the width of a cache line. Two, the potential marks was not worth disturbing the particle data structure and further altering the particle system. There is likely little benefit to ensuring the particles fit nicely inside a cache line as most of the CPU time is spent on OpenGL API calls. See chapter 2 ”Performance Mode” for more information.

1.2 Building

As of writing this report, I have yet to build and test on Windows. The Visual Studio project will build without issues, however, since this assignment was primarily designed and tested on Debain 12 ”Bookworm” (Linux 6.1.0-6-amd64) using AMD/Intel hardware (Mesa 22.3.6), I reccomend using CMake.

1.2.1 Caveats

The assignment makes use of a non-standard OpenGL extention during texture loading. "GL_TEXTURE_MAX_ANISOTROPY_EXT" should work on all modern Intel/AMD/Nvidia hardware, if it doesn't work on your hardware consider removing the line from texture.h and high_perf.cpp

1.2.2 Build Commands

```
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ../
make -j 16
./assign3
```

Figure 1.1: Linux build commands.

1.3 Usage

Keybindings and usage instructions are printed at program startup.

Performance Mode

2.1 Design

The high performance mode is the result of a weekend hack-a-ton where I wanted to see how easy it would be to implement a million particle+ renderer. If I had more time I would encapsulate the high_perf.cpp file into the particle system class, allowing for multiple *and customizable* particle systems. If you wish to change settings, most are constants in shaders/physics.comp or high_perf/high_perf.cpp. The rendering engine itself can handle around 20 million particles at about 60fps (Figure 2.2). With phyiscs enabled, the engine can handle about 6 million particles (Figure 2.4) but as Figure 2.5 shows, the renderer is clearly fillrate limited. Solutions to increase the number of rendered particles are discussed in section 2.5. It should be noted that (Figure 2.2) used a previous renderer which made use of a instanced "GL_TRIANGLES" approach and did not have textures or billboarding. The new renderer (Figure 2.3) makes use of "GL_POINTS" with a geometry shader to generate the vertices and features billboarding/texturing. A compute shader is used before rendering to update the particle positions and directions on the GPU. This way there is no need to copy the data to and from the graphics card.

2.2 Renderer

The legacy OpenGL renderer uses display lists to speedup rendering of the particles. Although this method is faster than using the same draw commands inline, it is highly limited by driver overhead. Modern GPUs are deisgned to process massive amounts of data all at once and benfit from reducing the amount of synchronization between the GPU and CPU. As mentioned earlier the current renderer uses an vertex buffer object to store all particle positions and directions in one giant array. It then uses those points to render all particles in a single draw call, thereby reducing driver overhead.

2.2.1 Rendering Pipeline

Vertex Shader

The vertex shader is purely used to passthough the particle position to the geometry shader. Since the vertex shader does not have the ability to output multiple vertices (not easily at least), we have to use a geometry shader.

Geometry Shader

The geometry shader uses the up and right vectors from the inverse view matrix to generate a quad facing the camera. It takes in the particle position and outputs a triangle strip. This is a highly efficent opperation as according to AMD there is dedicated hardware to handle this particular geometry shader case[4, p. 9].

Fragment Shader

The fragment shader is run once per pixel and is responsible for texturing the particles. I use a texture array as it can be bound once before rendering, therefore particles do not need to be sperated by texture. Using an array has the downside of every texture needs to be the same size, to solve this I resize the texture as it is loaded. Unforunately this will lead to some textures being distorted but the performance gain is worth it. The modern renderer is constrained by the lack of 'advanced' programming techniques, some of which are discussed in section 2.5.

2.3 Compute Shader

Compute shaders are very useful for embaressingly parallel tasks like updating particles. The compute shader is a very simple (nearly 1:1) translation of the CPU version of the particle system's update function. It handles 'dead' particles by reseting them to the inital position / direction. As a result particles are intialized with a random lifetime between 0 and the max lifetime to ensure even distribution. If you change the particle lifetime, please modify both constants!

Direction Offseting

Because generating random numbers on the GPU is hard (there is no dedicated hardware random number generator), I generate a random set of offsets at startup and upload these randoms to the GPU. The particle index is then used to access this buffer when the particle is reset; the result is a convincing distribution of particles. The large the number of particles the larger the offset buffer should be. Up to 6 million 8192 should be fine. If things look off consider increasing the value to some larger power of two. Make sure you update both constants here as well!

2.4 Usage

2.4.1 Building

Add ”-DEXTRAS=ON” to the CMake command.

```
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release -DEXTRAS=ON ../
make -j 16
./assign3
```

Figure 2.1: Linux build commands.

2.4.2 Running

All particles exist from the begining, which means all particles start at the inital position and slowly spread out. After starting the program but before moving around, you should press ’p’ to allow the compute shader to run, once the particles spread out it is safe to move. The slow performance of all the particles in the same spot has to do with overdraw accessing and writing the same location of the depth texture (hard to do in parallel). Fillrate is a common issue with this particle renderer. See the future plans section for possible resolutions.

2.5 Future Plans

Unfortunately because this is exam season, I do not have time to do anything more with this assignment. Furthermore I do not think the effort I’ve put in so far will be reflected in the value of the mark and any further improvements would be a waste. Below is a list of features I began looking into but as I have no experience with, they would require far too much experimentation and research to implement myself in a reasonable amount of time. As it is it took a weekend to implement something I was already somewhat familar with.

2.5.1 Lists

I would like to make it so all particles are not rendered all the time. Basically add a dead / alive particles list. This would prevent the issue of all particles starting in the same place, the low performance that causes and would be helpful in sorting.

2.5.2 Bitonic Sort

Professor Robson spent a good deal of time on this algorithmn in the parallel computing class and most of the literature online suggest using this, including the famous GDC talk on optimized particle systems[5]. The next step in implementing a good GPU accelerated particle system is bitonic sorting, however as I got further into the algorithmn it became clear that if I wanted to implement it myself, properly understanding the algorthmn would take too much time.

2.5.3 Tiling

Very similar to the rendering technique known as clustering, basically by dividing the view frustum into small sections and sorting / culling particles within we can reduce rendering load. This is an algothmn I've always wanted to implement, but lack a solid understanding of. The GDC talk goes further into this and many online resources talk about light clustering[5]. Again, if I had more time I would've learned it for this assignment as I think that would have been really cool.

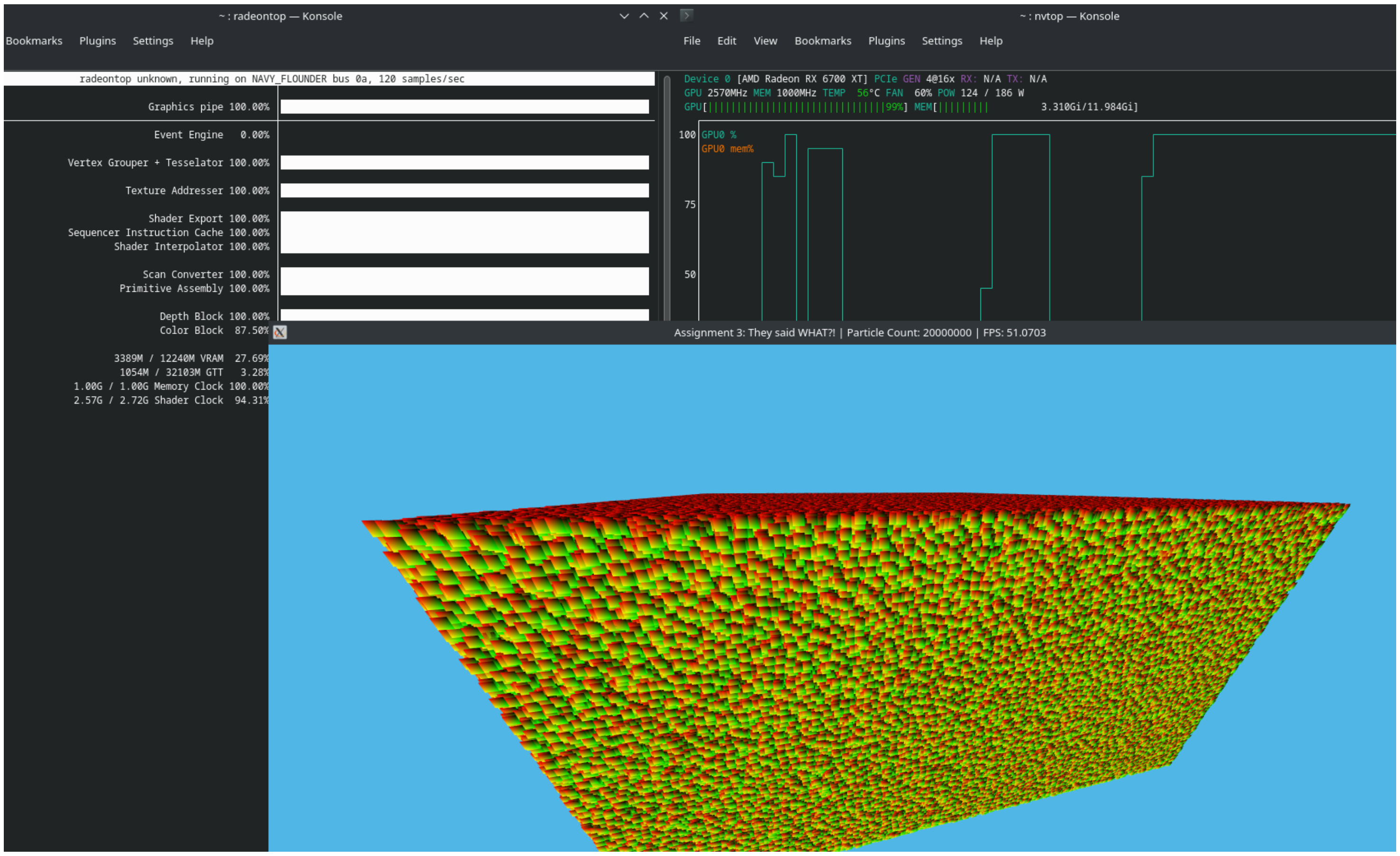
2.5.4 Hierarchical Depth Buffers

The idea is that by generating mipmaps of the depth buffer we can do broad phase culling of particles, thereby reducing the number of fine grained (per pixel) accesses to the depth buffer. This is a micro-optimization as stated by Mike Turitzin, but "every ms counts"[6].

2.5.5 Lighting

Since tiling particles comes from Forward+ rendering, it would make sense to implement a forward+ renderer.

2.6 Figures


~ : radeontop — Konsole
Bookmarks Plugins Settings Help
~ : nvtop — Konsole
File Edit View Bookmarks Plugins Settings Help
Assignment 3: They said WHAT?! | Particle Count: 20000000 | FPS: 51.0703

```
radeontop unknown, running on NAVY_FLOUNDER bus 0a, 120 samples/sec

                Graphics pipe 100.00%

                 Event Engine   0.00%

  Vertex Grouper + Tesselator 100.00%

            Texture Addresser 100.00%

                Shader Export 100.00%
Sequencer Instruction Cache 100.00%
           Shader Interpolator 100.00%

               Scan Converter 100.00%
           Primitive Assembly 100.00%

                  Depth Block 100.00%
                  Color Block  87.50%

         3389M / 12240M VRAM  27.69%
          1054M / 32103M GTT   3.28%
   1.00G / 1.00G Memory Clock 100.00%
   2.57G / 2.72G Shader Clock  94.31%
```

```
Device 0 [AMD Radeon RX 6700 XT] PCIe GEN 4@16x RX: N/A TX: N/A
GPU 2570MHz MEM 1000MHz TEMP  56°C FAN  60% POW 124 / 186 W
GPU[||||||||||||||||||||||||||||||||99%] MEM[|||||||||         3.310Gi/11.984Gi]

100 GPU0 %
    GPU0 mem%

 75

 50
```

Figure 2.2: 20 million particles distributed in a 50x25x50 cube with load monitors

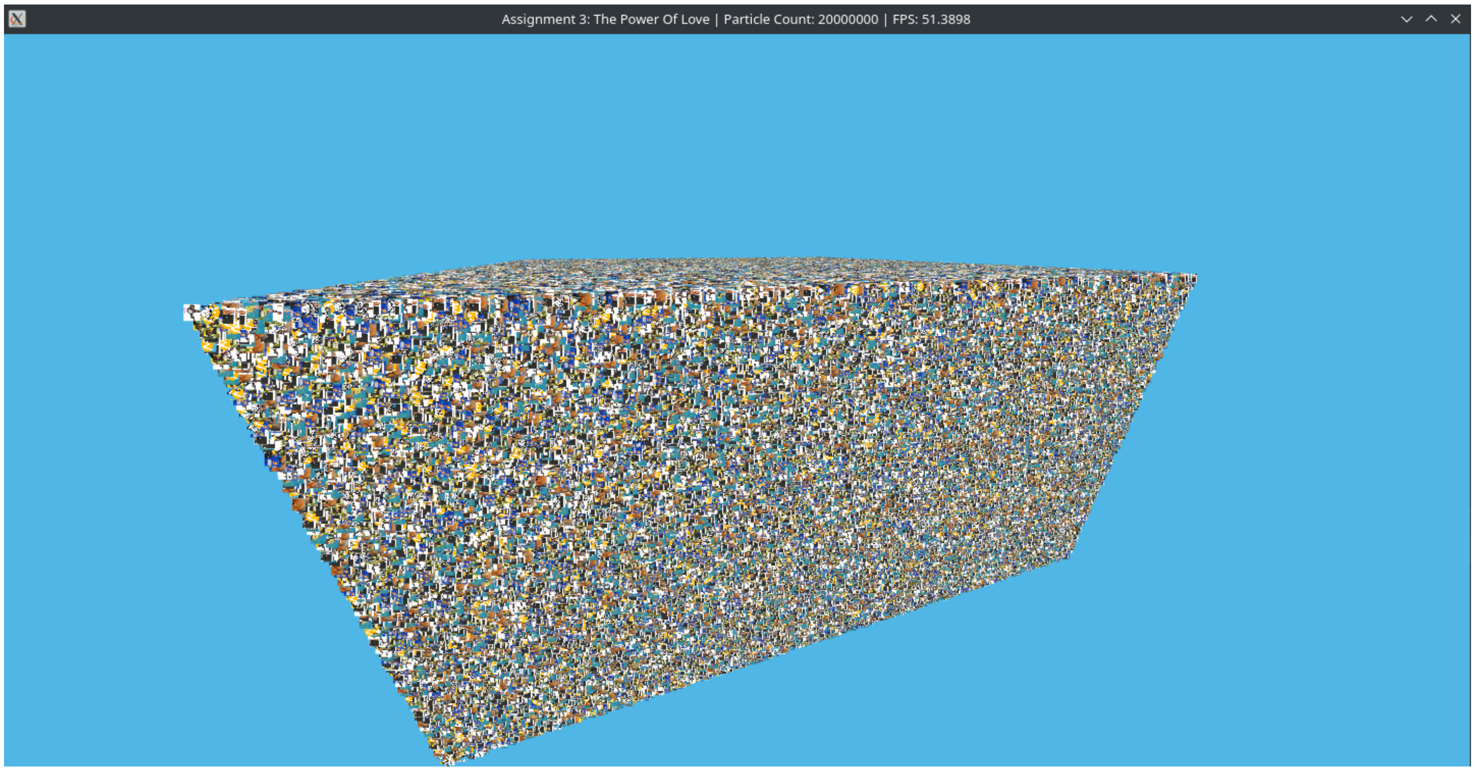

Assignment 3: The Power Of Love | Particle Count: 20000000 | FPS: 51.3898

Figure 2.3: 20 million particles on the new renderer

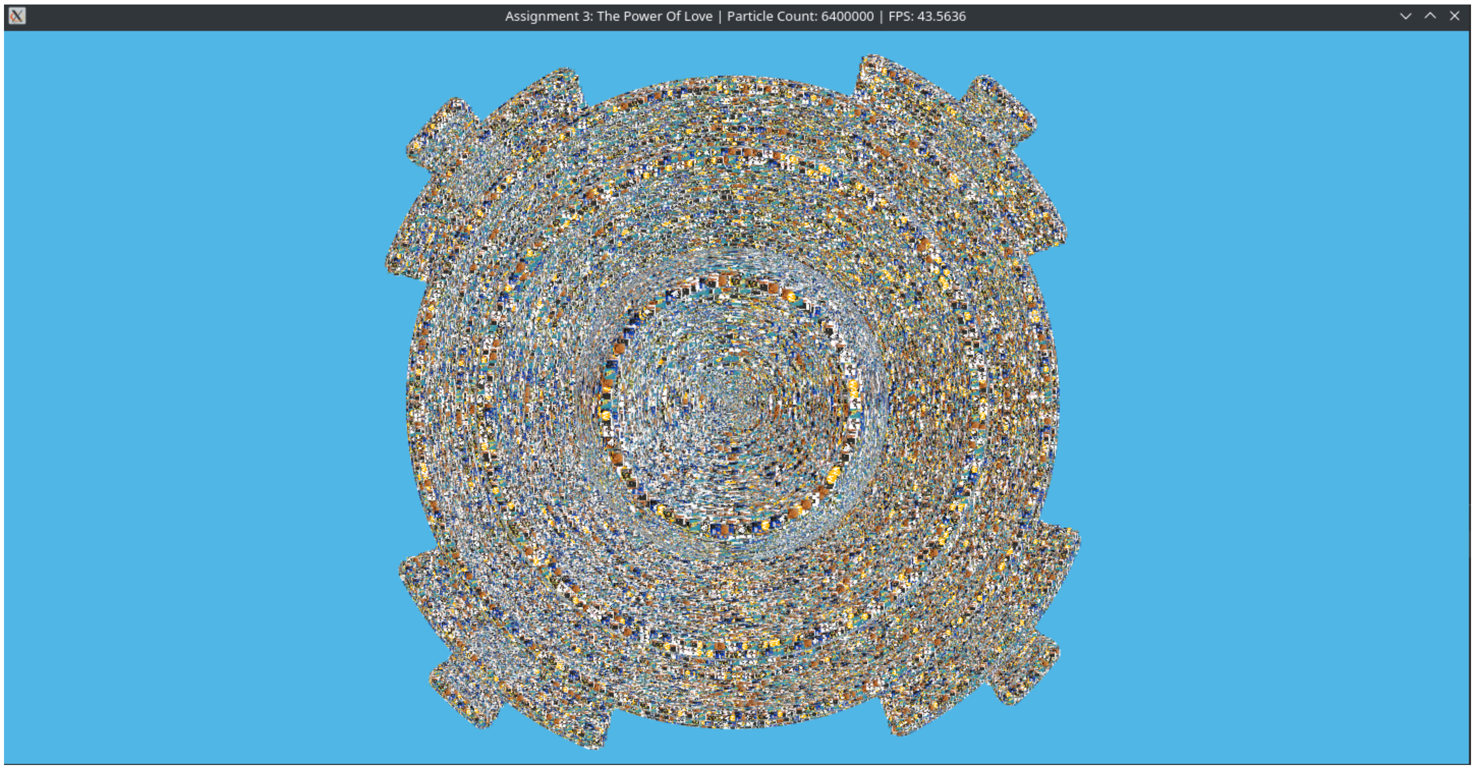

Assignment 3: The Power Of Love | Particle Count: 6400000 | FPS: 43.5636

Figure 2.4: 6.4 million particles, fillrate (not compute) limited.

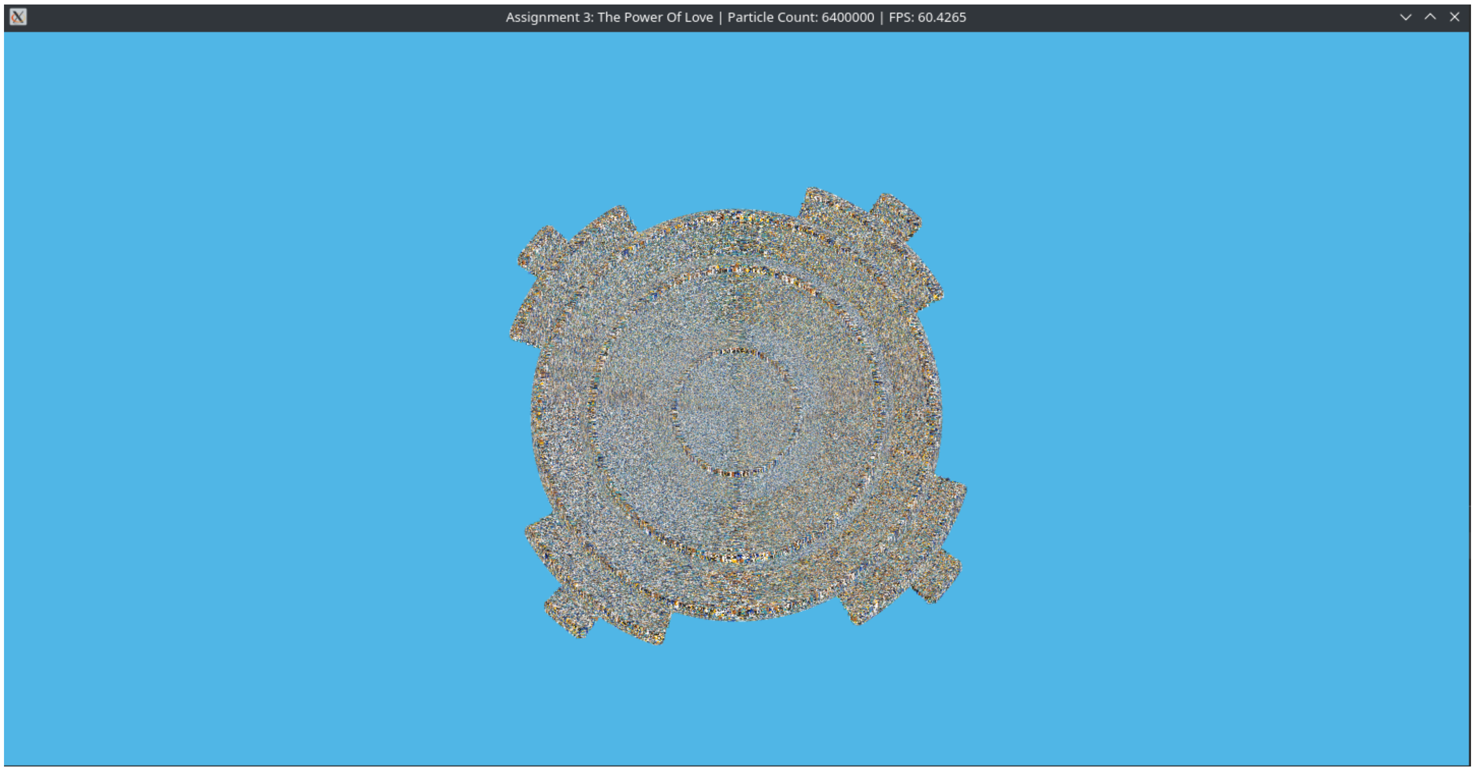

Assignment 3: The Power Of Love | Particle Count: 6400000 | FPS: 60.4265

Figure 2.5: 6.4 million particles, zoomed out, showing fillrate as the limiting factor in speed.

Bibliography

[1] Mike Bailey. OpenGL Compute Shaders. "https://web.engr.oregonstate.edu/~mjb/cs519/Handouts/compute.shader.2pp.pdf", 2021.

[2] Krnonos Group. OpenGL Reference Manual. "https://registry.khronos.org/OpenGL-Refpages/gl4/", 2014.

[3] JeGX. Particle Billboarding with the Geometry Shader. "https://www.geeks3d.com/20140815/particle-billboarding-with-the-geometry-shader-glsl/", 2014.

[4] Emil Persson. ATI Radeon HD 2000 programming guide. "https://web.archive.org/web/20160722164341/http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/ATI_Radeon_HD_2000_programming_guide.pdf", 2007.

[5] Gareth Thomas. Compute-Based GPU Particle Systems. "https://ubm-twvideo01.s3.amazonaws.com/o1/vault/GDC2014/Presentations/Gareth_Thomas_Compute-based_GPU_Particle.pdf", 2014.

[6] Mike Turitzin. Hierarchical Depth Buffers. "https://miketuritzin.com/post/hierarchical-depth-buffers/", 2020.