***Direct X Homework Day 8:***

***Compute Shader Particles***

**Objective**

Today’s homework we’ll be creating a particle system using the geometry shader, and compute shader.

**Homework Requirements**

Features you’re expected to implement:

* Make a working particle system that is handled by the compute shader.

**Homework Instructions**

**Particles**

* By this point in the program, you’ve had to deal with particles at least once. Today’s homework has you making one of your own within the compute shader, and possibly (if you choose to) the geometry shader.
* A particle is just a single point that is modified over time in one or more ways, whether it be color, movement or something else. There are some slides in Wendy’s old material that cover creating a particle system code side, but some of that doesn’t apply here (for loops for instance) due to the parallel nature of the compute shader. The basic logic is as follows:
* If a particle’s **current lifespan** is less than its **maximum life span**
  + Increment the **current lifespan** by **delta time**
  + We can then get a **0-1 ratio** of how much **life remains** for the particle
  + We can now use our ratio as a modifier for any of our variables that have **start** and **end** values (For instance, start color and end color) by **lerping** between the two values.
  + Here is also where we’ll apply the **velocity** to the particle’s **position**.
  + Otherwise, If our **thread id** is less than our **number of particles** (meaning it should be a living particle), randomize its velocities, position, lifespan, set its current life to 0,  and set its current color to start color.
  + Particles will need at least the following things:
    - Position
    - Color
    - Velocity vector
    - Current life
    - Max life
    - The particle system itself will need to have:
      * Start Color
      * End Color
      * Min Velocity
      * Max Velocity
      * Number of Particles
      * Spawn rate (The amount you can spawn per frame)
      * Min Life
      * Max Life
      * Min Starting Position
      * Max Starting Position
      * Delta Time
      * Total Time (the time since the program started, used by our random number system)

**Tornado**

* In the example, we’re creating a tornado by creating particles that rotate and move upwards. The rotation is done simply by creating a **Y rotation matrix** with a rotation value of **(deltaTime \*20\*life remaining ratio % Two Pi)**. This will create a speed that increases the longer the particle has existed.
* We’ll modify our **velocity vector’s Y**value by subtracting **(delta time\* 3 \* inverse of our life remaining ratio)** from it to cause our tornado to flatten out towards the top.
* We’ll then rotate our **velocity vector** by our **rotation matrix** (so the position keeps rotating when it’s updated).
* Lerp between the start color and end color.
* Finally, all we need to do is multiply our velocity by delta time, and add it into our particle’s position.

**Geometry Shader**

* It’s not necessary to use the geometry shader for this lab, but man will it make your life easier. One tip is if you do use the geometry shader to create the textured quads needed by the particles, you’ll need them to face the screen. You can do this by creating the quads in view or projection space.
* If you decide to do this to keep your quads from being rectangles, you’ll need to modify the verts by the aspect ratio.

**Pseudo-random behavior**

* One of the things we’ll want to do in our shader is implement pseudo-random behavior for our particles. Even with all of DirectX 11’s cool new features, HLSL is still missing a function for number randomization.  Thankfully, we’ve got a simple example of one you can use for your shaders:

// compute a random floating point value based on index and time seed data

// credit for the hashing code goes to Robert Jenkins.

// Not random to the level of say a "Mersenne Twister" but visually acceptable.

float PsuedoRandF(float min, float max, uint seedIndex, float timeOffset)

{

            // mix the seed index with a Robert Jenkins 32bit interger hash.

            seedIndex = (seedIndex+0x7ed55d16) + (seedIndex<<12);

            seedIndex = (seedIndex^0xc761c23c) ^ (seedIndex>>19);

            seedIndex = (seedIndex+0x165667b1) + (seedIndex<<5);

            seedIndex = (seedIndex+0xd3a2646c) ^ (seedIndex<<9);

            seedIndex = (seedIndex+0xfd7046c5) + (seedIndex<<3);

            seedIndex = (seedIndex^0xb55a4f09) ^ (seedIndex>>16);

            // offset our mixed seed by the timeOffset and mix one last time

            seedIndex += asuint(timeOffset);

            // second mix, once again using the Jenkins hash.

            seedIndex = (seedIndex+0x7ed55d16) + (seedIndex<<12);

            seedIndex = (seedIndex^0xc761c23c) ^ (seedIndex>>19);

            seedIndex = (seedIndex+0x165667b1) + (seedIndex<<5);

            seedIndex = (seedIndex+0xd3a2646c) ^ (seedIndex<<9);

            seedIndex = (seedIndex+0xfd7046c5) + (seedIndex<<3);

            seedIndex = (seedIndex^0xb55a4f09) ^ (seedIndex>>16);

            // with our bits avalanched sufficiently we can convert to a floating point range.

            return (seedIndex/float(0xFFFFFFFFu)) \* (max-min) + min;

}

**Packing Order**

* This is **HUGE** and can save some headaches in the future. Here’s Lari’s comment from his code to explain it:

// PLEASE READ THE BELOW TO UNDERSTAND HOW TO MAKE HLSL AND C++ STRUCTURES PACK IDENTICALLY

////////////////////////////////////////////////////////////////////////////////////////////

// HLSL packing rules are similar to performing a #pragma pack 4 with Visual Studio,

// which packs data into 4-byte boundaries. Additionally, HLSL packs data so that

// it does not cross a 16-byte boundary. Variables are packed into a given four-component

// vector until the variable will straddle a 4-vector boundary; the next variables will be

// bounced to the next four-component vector.

//

// Each structure forces the next variable to start on the next four-component vector.

// This sometimes generates padding for arrays of structures. The resulting size of any

// structure will always be evenly divisible by sizeof(four-component vector).

////////////////////////////////////////////////////////////////////////////////////////////