Flood (Working Title) Midterm Report

I have worked with Unity for several years, initially diving into it during high school after experimenting with Pygame and game modding. I have taught myself a considerable amount about the program, with this being my first formal course on the matter. Most of my work started with random ideas for a project, supported by dozens of browser tabs full of Unity's documentation. An idea for shared initialized objects to prevent redundant initializations led me to object pooling. An idea for more convenient integration with MonoBehaviours led me to the singleton and observer design patterns. An idea for stereoscopic vision led me to connected-component labeling, and then my GPU-based implementation of my algorithm led me to compute shaders. General curiosity had me up late at night trying to master concepts I didn't even know the formal names of, but mountains of documentation and Stack Overflow pages helped me learn what I know today.

Our group's project is tentatively titled, *Flood*, and came from a concept I had been toying with in my head for about a month prior to the start of the semester. The game is a freeform strategic survival game where the only enemy is an ever-spreading simulated fluid. The player must develop logistics networks to collect and carry resources in an attempt to halt (and eventually repel) this encroaching fluid. My work this semester has primarily been towards the conceptualization of the game, as well as the core fluid system that the game is built around. Careful design of this system will allow for simple addition of surface-level mechanics and components later in development. Elaborate systems of turrets, barriers, pipes, power distribution, and

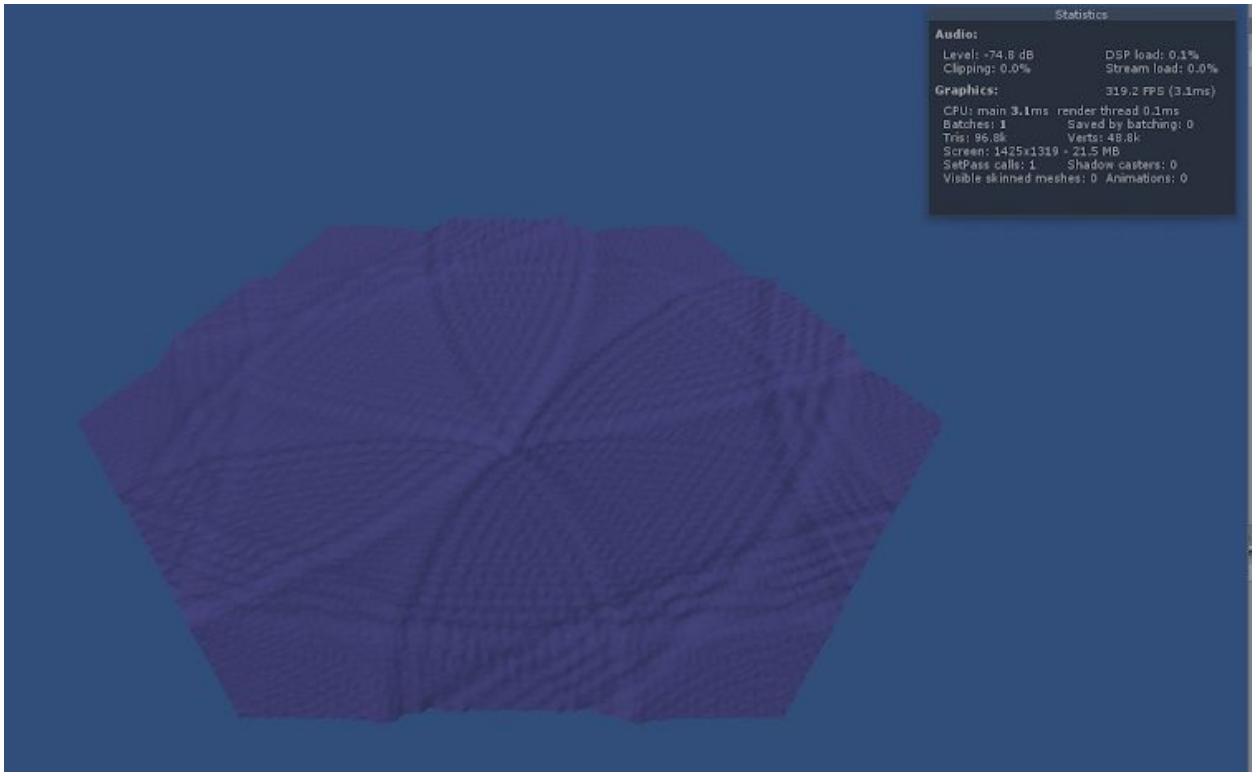
circuit logic compose some of the feature ideas I've contributed for the project. In addition, I have currently written code for most of the core systems in the game, including the fluid system, terrain system, managers, mesh and hexagonal-grid helper libraries, and some additional shaders and utility classes.

All of the assets I've created so far have been simple shells for procedurally-generated content. Incidentally, this is the area that I am currently working on. I am experimenting with several fluid algorithms that I have come up with, trying to determine which will work best with the given requirements. This needs to be a system that is computationally-inexpensive and deterministic, while producing fluid that can flow through terrain and be cheaply created or destroyed arbitrarily. Most research I have done for usable algorithms has only yielded systems for use in advanced computer graphics and scientific simulations, far from suitable for real-time applications. As a result, I have taken to developing my own, producing algorithms with varying strengths and weaknesses (and thus varying suitability for our project). Aside from the fluid simulator, a large portion of my work has been put towards the hexagonal-lattice structure I designed in order to allow for more realistic fluid flow behaviors. This grid structure is deceptively tricky to work with, as it cannot be intuitively mapped to a multidimensional array. This is due to the lattice having a non-linearly-independent spanning set of 3 coordinate vectors, an aspect derived from the 6 cardinal directions of hexagonal tessellation. My current solution is the development of layers of abstraction that map the 3-axis hexagon coordinates to an underlying 2-dimensional grid. This is

critical for both conversion to world-space, as well as creating a deterministic procedural-noise terrain generator that will cooperate with this hexagonal form.

```
RWTexture2D<float> Front;
RWTexture2D<float> Back;
RWTexture2D<float> Terrain;
RWTexture2D<float> Mask;

// int3(x, y, useOddOffset)
static const int3 NEIGHBOR_OFFSETS[7] = {
    int3(  0,  0,  0),
    int3(  0,  1,  0),
    int3( -1,  1,  1),
    int3( -1,  0,  1),
    int3(  0, -1,  0),
    int3(  1,  0,  1),
    int3(  1,  1,  1)
};

[numthreads(8, 8, 1)]
void Update(int3 id : SV_DispatchThreadID)
{
    if (!Mask[id.xy]) return;

    float C = 0.6f;

    float nCount = 0;
    float nTotal = 0;

    int hasNeighbor;
    int2 v;

    int2 oddOffset = int2(0, -(id.x & 1));

    // TODO: Cache neighbors into TGSM to minimize required memory bandwidth
    [unroll(6)] for (int n = 1; n <= 6; n++) {
        v = id.xy + NEIGHBOR_OFFSETS[n].xy + NEIGHBOR_OFFSETS[n].z * oddOffset;
        hasNeighbor = Mask[v]; // Needs terrain check
        nCount += hasNeighbor;
        nTotal += hasNeighbor * Back[v];
    }

    /* ... */

    if (nCount == 0) return;

    Front[id.xy] = Back[id.xy] + 0.995f * (Back[id.xy] - Front[id.xy]) + C * C * (nTotal - nCount * Back[id.xy]);
}
```

Current iteration of one of the fluid simulation algorithms


Statistics
Audio:
Level: -74.8 dB
DSP load: 0.1%
Clipping: 0.0%
Stream load: 0.0%
Graphics:
319.2 FPS (3.1ms)
CPU: main 3.1ms render thread 0.1ms
Batches: 1
Saved by batching: 0
Tris: 96.8k
Verts: 48.8k
Screen: 1425x1319 - 21.5 MB
SetPass calls: 1
Shadow casters: 0
Visible skinned meshes: 0
Animations: 0