Flood Final 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 *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 manage limited resources and plan a strategy to defend against, and eventually repel, the encroaching fluid. My work this semester has been on the game concept, as well as the programming of the game's core features and functionality. Careful design of this system will allow for simple addition of further mechanics and components later on if we decide to keep developing it.

This game is largely based on procedurally-generated content, so the only assets I've developed have been simple shell prefabs for the Generator systems. I had experimented with several fluid algorithms that I have come up with, trying to determine which will work best with the given requirements. This needed 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 had done for usable algorithms only yielded systems for use in advanced computer graphics and scientific simulations, far from suitable for real-time applications. As a result, I ended up developing my own, producing algorithms with varying strengths and weaknesses (and thus varying suitability for our project) before settling on one that would work best. Aside from the fluid simulator, was put towards the hexagonal-lattice structure I designed for the game world in order to allow for more realistic fluid flow behaviors, and the accompanying terrain generators and mesh generators that need to work with it. 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 solution was to develop the HexGrid class, which combined layers of abstraction that map the 3-axis hexagon coordinates to an underlying 2-dimensional grid. This was critical for both conversion to world-space, as well as creating a deterministic procedural-noise terrain generator that will cooperate with this hexagonal form. The abstraction similarly enabled more approachable additions and changes to game behaviors.

```
static const float THETA = 0.866025403784438646763723170752936183471402626905190314027f; // sqrt(3)/2
static const float3 NEIGHBOR_NORMAL_LATERALS[7] = {
    float3(   0, 1,      0),
    float3(  -1, 0,      0),
    float3(-0.5, 0,  THETA),
    float3( 0.5, 0,  THETA),
    float3(   1, 0,      0),
    float3( 0.5, 0, -THETA),
    float3(-0.5, 0, -THETA)
};

// 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)
};

void vert(inout appdata_full v) {
    #ifdef SHADER_API_D3D11

    int2 iVec = v.texcoord.xy; // UV is flipped for some reason

    float curVolume = _FluidMap[iVec.yx].r;
    float curHeight = curVolume + _TerrainMap[iVec.yx].r;

    int2 oddOffset = int2(0, -(iVec.y & 1));

    int hasNeighbor;
    int2 nVec;
    float nCount = 0;
    float nLowCount = 0;
    float nTotalHeight = 0;

    uint mask = _Mask[iVec.yx].r;

    // TODO: Cache neighbor values in TGSM
    [unroll(6)] for (int n = 1; n <= 6; n++) {
        nVec = iVec.yx + NEIGHBOR_OFFSETS[n].xy + NEIGHBOR_OFFSETS[n].z * oddOffset;
        hasNeighbor = ((mask & (1 << n)) > 0) && (_FluidMap[nVec].r > 0);
        nCount += hasNeighbor;
        hasNeighbor = hasNeighbor && ((_FluidMap[nVec].r + _TerrainMap[nVec].r) <= curHeight);
        nLowCount += hasNeighbor;
        nTotalHeight += hasNeighbor * (_FluidMap[nVec].r + _TerrainMap[nVec].r);
    }

    float offset = (curVolume > 0) ? curHeight : -1;

    v.normal = float3(0, 1, 0);
    float3 nTotalNormal = float3(0, 0, 0);
    nCount = 0;

    [unroll(6)] for (int n = 1; n <= 6; n++) { // TODO: Combine this with height pass
        nVec = iVec.yx + NEIGHBOR_OFFSETS[n].xy + NEIGHBOR_OFFSETS[n].z * oddOffset;
        hasNeighbor = ((mask & (1 << n)) > 0) && (_FluidMap[nVec].r > 0);
        nCount += hasNeighbor;
        nTotalNormal += hasNeighbor * normalize((_FluidMap[nVec].r + _TerrainMap[nVec].r - offset) * NEIGHBOR_NORMAL_LATERALS[n] + float3(0, 1, 0));
    }

    v.normal = nTotalNormal / nCount;
    v.vertex.y += offset;
```

Part of the fluid shader that handled vertex displacement and normals

```
public class HexGrid<T>
{
    // Even-row offset hexagonal grid

    // int3(x, y, useOddOffset)
    static readonly Vector3Int[] NEIGHBOR_OFFSETS = {
    new Vector3Int(  0,  0,  0),
    new Vector3Int(  1,  0,  0),
    new Vector3Int(  1, -1,  1),
    new Vector3Int(  0, -1,  1),
    new Vector3Int( -1,  0,  0),
    new Vector3Int(  0,  1,  1),
    new Vector3Int(  1,  1,  1)
};


    private T[,] grid;

    public int Size { get; set; } // Side length


    public HexGrid(int size){
        Size = size;

        grid = new T[Size, Size];
    }

    public T this[int x, int y] {
        get { return grid[x, y]; }
        set { grid[x, y] = value; }
    }

    public T this[Vector2Int v] {
        get { return this[v.x, v.y]; }
        set { this[v.x, v.y] = value; }
    }

    public T GetNeighbor(int x, int y, int n) {
        Vector2Int nVec = NeighborCoords(x, y, n);
        return this[nVec.x, nVec.y];
    }

    public void SetNeighbor(int x, int y, int n, T val) {
        this[NeighborCoords(x, y, n)] = val;
    }

    public bool HasNeighbor(int x, int y, int n) {
        Vector2Int nVec = NeighborCoords(x, y, n);
        return (nVec.x >= 0 && nVec.x < Size) && (nVec.y >= 0 && nVec.y < Size);
    }

    private Vector2Int NeighborCoords(int x, int y, int n) {
        Vector3Int nOffset = NEIGHBOR_OFFSETS[n];
        int oddOffset = -(y & 1);

        return new Vector2Int(x + nOffset.x + nOffset.z * oddOffset, y + nOffset.y);
    }

}
```

The HexGrid class represents a 2D array with abstraction for Hexagonal Coordinates