CG Final Report - Water Rendering

B11902045 黃泓予 / B11902083 徐粲桓 / B11902050 范綱佑

Abstract

The aim of this project is to study a technique for fluid simulation called Smoothed-particle hydrodynamics or SPH. The project is splitted into two main parts, paper survey and implementation.

Implementation and Execution

We implement SPH-based fluid simulation in Unity 6.1 (6000.1.1f1). The code has been tested functional on Nvidia and Apple M-series GPU.

The source code of our implementation can be seen here: https://github.com/s1imlix/cg-final

Structure

Here, we list the files in `Asset`, which define most important behaviours of our program, its structure is as follows with metadata omitted:

```
Assets
├── Materials
|   ├── Sphere.obj
|   ├── WaterMaterial.mat
|   ├── WaterMaterialMarching.mat
|   └── WaterMaterialMC.mat
├── ParticleController.cs
├── Resources
|   ├── BitonicMergeSort.compute
|   ├── MarchingCube.compute
|   ├── MarchingCubeHelpers.hlsl
|   └── MarchingCubeLUT.txt
├── Scenes
|   └── SampleScene.unity
├── Scripts
|   ├── Fluid3D
|   |   ├── Compute
|   |   |   ├── GPUSort.cs
|   |   |   ├── SmoothingKernel.hlsl
|   |   |   ├── SpatialHash.hlsl
|   |   |   └── SPHCompute.compute
|   |   ├── Shader
|   |   |   ├── MarchingCubeMesh.shader
|   |   |   ├── MarchingCubeRenderer.cs
|   |   |   ├── ParticleRenderer.cs
|   |   |   └── ParticleSphere.shader
|   |   └── SPH.cs
|   ├── Helpers
|   |   └── ComputeHelper.cs
```

SPH

We implemented a Smoothed Particle Hydrodynamics (SPH) simulation system in Unity using compute shaders for GPU acceleration. Core logic of the system reside in `SPH.cs`, serving purposes of initializations of buffers and individual particles.

The SPH system, as described in the Survey section, computes the following forces for each particle: External Force (in the current scenario, Gravity), Pressure Force, Viscosity Force and Collision Damping.

The actual computation of the aforementioned variable is dispatched by the `SPH.cs` to the GPU thread groups with calculation defined in `SPHCompute.compute` according to the referred paper.

Each particle is defined with its position, velocity, predicted position, density, and near-density. Within the program, we employed several buffer to facilitate transferring data between CPU and GPU. Namely, `_ParticleBuffer`, `_spatialLookupBuffer` and `_startIndicesBuffer`.

- `_ParticleBuffer` simply stores per-particle data.
- `_spatialLookupBuffer` and `_startIndicesBuffer` are related to a fast neighbor-lookup algorithm, which we will explain in the next section.

Fast-lookup

As described in the paper survey, the scalar updates of each particles involves summing over the same scalar on nearby particles within some radius r. Naive approach of checking all particle's distance is computationally expensive by $O(n^2)$. Thus, we implemented a fast-lookup mechanism that can quickly list of particles within distance r.

The idea is to split the space into cells (these cells are different from the ones in marching cube) of side length r, then the relevant particles are within the $(\pm1, \pm1, \pm1)$ offset from the current cell, 27 cells at most.

Then, we define $h(x, y, z) = K_1 \times x + K_2 \times y + K_3 \times z$ to be hashed value of a cell. Using h we can map list of particle positions into list of hashes. By sorting (with Bitonic Merge sort, implemented in `GPUSort.cs` but we omit the details here) the particles with their hashes as key, we will have particles in the same cell aligned together -> `_spatialLookupBuffer`

Now, running a kernel over sorted entries and determine the boundaries of different hashes allow us to look up a hash value and get its starting index in `_spatialLookupBuffer` -> `_startIndicesBuffer`.

Finally, h can produce collision so we need to keep their actual hashes inside entries of `_spatialLookupBuffer`.

The pseudo code for the full procedure during particle update would then be:

```
  cellID = PosToCell(particle.pos)

  for (i,j,k) enumerated over (-1,0,1)
      cellHash = h(cellID.x + i, cellID.y + j, cellID.z + k)
      cellKey = cellHash % n
      curr = _startIndicesBuffer[cellKey]
      while (curr < n)
          curr_particle = _spatialLookupBuffer[curr]
          if (curr_particle.key != cellKey) break;//current cell is done
          if (curr_particle.hash != cellHash) continue;//h collision
          accumulate curr_particle scalar data
```

Marching Cube

Marching Cube is an algorithm that converts a 3D density field (like fluid simulation data) into a triangulated mesh surface by finding where the density crosses an "iso-value" threshold.

In essence, the flow of our Marching Cube can be described as follow:

1. SPH system generates 3D density texture from particle data: Since Marching cube is based on sampling density, the algorithm depends on the density texture, updated in `SPH.cs`, to work.
2. Marching cubes compute shader processes density field: Here, core logic of the algorithm, defined in `MarchCube` of `MarchingCube.compute` is performed. As each time we update, we first extract density values at 8 cube corners. Depending on the combinations, we use the Look Up Table to decide which triangle(s) pattern to render. Then, to provide a precise and realistic look, we interpolate the vertices to calculate exact surface intersection points instead of uniformly setting them to the midpoint of the given edge. Finally, we compute surface normals using density gradients and store above results to the `triangleBuffer`.
3. Triangles rendered as mesh using indirect drawing: With the triangle calculated, we copy the informations to a `RenderArgs` and use `Graphics.DrawProceduralIndirect(marchingCubeMaterial, bounds, MeshTopology.Triangles, renderArgs);` to actually draw the triangles
4. Process repeats each frame

`MarchingCubeRenderer.cs`

This file implements the rendering component of the SPH simulation using the Marching Cubes algorithm to generate a triangle mesh from a 3D scalar density field. It also define vertex (position and normal) and triangle structs (3 vertices) for GPU output.

`MarchingCube.compute`

This file determines the surface geometry by evaluating total of 8 vertices of a cube in the 3D density field built up by particle mentioned in SPH section. For each vertex, it computes an 8-bit index based on whether each corner's density is above or below the specified isovalue threshold. This 8-bit value (ranging from 0 to 255) is then used to look up a precomputed edge table that specifies which edges are intersected by the surface and how triangles should be formed. This lookup-driven approach allows efficient generation of surface triangles in parallel on the GPU.

Paper Survey

We surveyed Müller, M., Charypar, D., & Gross, M. (2003). Particle-Based Fluid Simulation for Interactive Applications. In Proceedings of the Eurographics/SIGGRAPH Symposium on Computer Animation (pp. 154–159). Eurographics Association.

This is a paper studying on fluid simulation with SPH. We provide a summary of mathematical details behind SPH and how SPH-simulated particles are rendered into fluid. On top of that, we also surveys some recent work citing this paper and discuss their innovations.

SPH

The main idea behind our project is SPH, a computational method for simulating continuous medium (a solid or a body of fluid) SPH interpolates field quantities defined in discrete location (the particles) and evaluates continuously in space.

In SPH, to evaluate a scalar quantity A at location $\mathbf{r}$ is defined by weighted sum of that quantity A_j, from other particle j **within radius** h.

$A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h)$ where:

- m_j: mass of particle j
- ρ_j: density at j
- A_j: scalar value at j
- $\mathbf{r}_j$: position of particle j

The mass m_i and density ρ_j defines a per-particle volume $V_i = m_i/\rho_j$, to evaluates density $\rho_S(\mathbf{r})$ at position $\mathbf{r}$ now, we may use:

$\rho_S(\mathbf{r}) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h)$

We may also obtain the differential operators by simply differentiate them, should it be needed:

- Gradient of a Field A: $\nabla A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h)$
- Laplacian of a Field A: $\nabla^2 A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h)$

Smoothing Kernel

In SPH, physical quantities like pressure, density, and velocity are only defined at particle positions. To evaluate these quantities at arbitrary locations in space (e.g., for computing gradients or visualizing flow), we need a way to "spread" the particle data smoothly over space.

The smoothing kernel $W(\mathbf{r}, h)$ acts as a weighting function that distributes a particle's influence over a finite neighborhood (within core radius h).

To ensure second-order interpolation accuracy, it must be:

- Even: $W(\mathbf{r}, h) = W(-\mathbf{r}, h)$
- Normalized: $\int W(\mathbf{r}) d\mathbf{r} = 1$ These properties ensure second-order interpolation accuracy.

Spatial hash

When SPH calculates interactions between particles, the computational complexity is typically $O(n^2)$, which becomes inefficient when simulating thousands of particles. However, by using a spatial hash, only particles within a radius-r cell are considered as mentioned in previous `Fast-lookup` section. This optimization reduces the complexity to approximately $O(nm)$, where m is the average number of particles per grid cell.

Workload

- B11902045 黃泓予：Report/Presentation
- B11902083 徐粲桓：Marching Cube, SPH
- B11902050 范綱佑：Parts of SPH, Report/Presentation

References

Particle-Based Fluid Simulation for Interactive Applications

Coding Adventures - Simulating Fluids

Coding Adventures - Rendering Fluids

Coding a Realtime Fluid