WebGL based 3D Snow Simulation - Milestone Report

Fan Zhang*
EECS Department
University of California, Berkeley
fan.zhang@berkeley.edu

Yu-chieh Lan*
EECS Department
University of California, Berkeley
yclan2@berkeley.edu

Zhaoxiong Cui*
EECS Department
University of California, Berkeley
zhaoxiong.cui@berkeley.edu

1 INTRODUCTION

Snow is a common natural phenomenon in the real world which covers the ground all the white in the winter. However, in computer graphics, it is a challenging task to model the snow digitally. Different from other weather phenomena, snow has both the solid-like and fluid-like properties which are difficult to handle. In the meantime, the dynamic scenes of snow can be divided into three parts, namely, snow falling, snow accumulation, snow wind interaction. What we need is not only to render a unique texture of snow, but also to recover the behavior of snow particles. Since snow is composed of small ice particles, it is a granular material, so we can representation is with the particle system [7]. Luckily, researchers have figured out the way to model the snow as a particle system [6].

Currently, most of the computer graphics works are implemented in C++ with OpenGL since it supports GPU. However, to render a demo in OpenGL, we need to download the source code, compile it and then render it, which is complicated. Inspired by a really cool WebGL demo, we want to implement the snow simulation based on WebGL [5]. Our goal is to construct snowy scenes and accomplish the animation simulation in real-time. The scenes could include different shapes of snow on different terrains. Because it is a web page, we should also include a user interface that enables users to design and interact with the scene. Tuning bars could also be provided on the interface to adjust some parameters within reasonable ranges. Real-time interaction may be an interesting function, users can catch the falling snow by dragging different containers.

2 CURRENT PROGRESS

We divided our project into three most important parts: Shader, Physical Property and Web Interaction. We developed these three parts simultaneously and decided to merge them into a single web application in the end since since it is easy for us to debug them separately. The following parts will give a detailed explaination on what we have done.

2.1 WebGL Framework

We developed the WebGL framework based on Three.js [4] which is a cross-browser JavaScript library and Application Programming Interface (API) used to create and display animated 3D computer graphics in a web browser. Based on it, we build our start javascript scene which contains a flower, an island and the light illuminating. Then, for the snow particles, we constructed some data structures to represent the basic transforms and shading effect in figure 1.

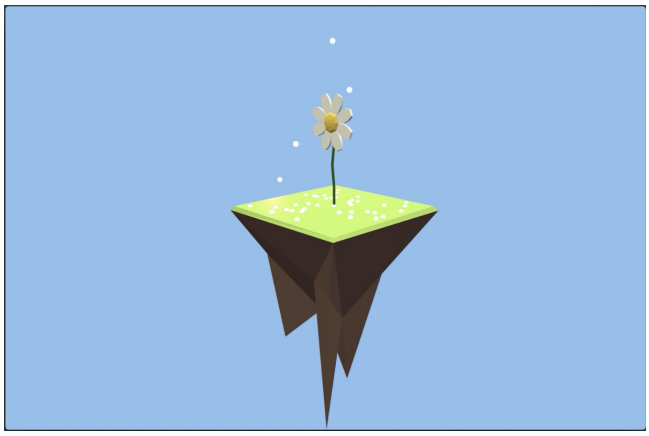

Figure 1: The basic scene to render the snow falls in an island

Figure 2 shows the code structure of our start scene. Every kind of object is represented as a **.js* file under the SeedScene class. Currently we only implemented the basic function of a single particle generated from random position and falls into the surface of island.

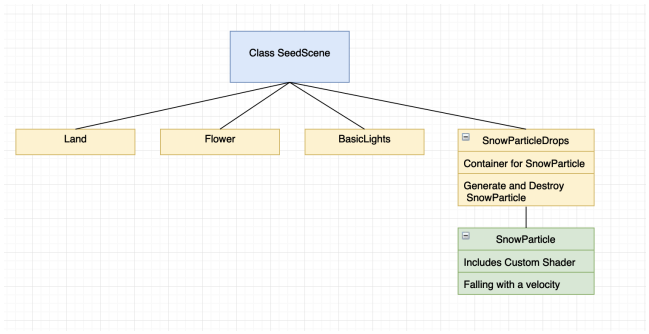

Figure 2: The code structure based on Three.js

2.2 Core Data Structure

To facilitate the three-parts work flow, we designed several key data structures as a bridge between the physical modeling (simulation) and the rendering pipeline.

For the Physical part, as described in [6], the snow is represented as a collection of particles. Each particle holds its position $\mathbf{p}$, velocity $\mathbf{v}$, and mass m. In each step of iteration, these physical properties will evolve according to the physical process defined in [6]. In the middle of each step, some additional data structures, like a grid that divides the space dimensions and grouping the mass points

together. Yet, the three properties remain sufficient to describe the system at the beginning and the end of each step.

For the rendering pipeline, the goal is to render a realistic view of snow, regardless how the status of snow is determined. Speaking with the three properties of physics, this rendering relies only on the positions of each snow particles in a static frame. The rendering also requires some geometry properties of the particles, like its radius (if we assume each particle has sphere meshes). Plus, we will need a

Based on above analyses, we figured out a clear interface design which will seamlessly integrate the physical simulation and the rendering pipeline together. The core of this design is a data structure representing a snow particle that will hold properties sufficient for both the physical and the rendering tasks.

Listing 1 shows a simplified definition of the `SnowParticle` class. At the top level, it inherits `THREE.Object3D`. This inheritance makes itself "renderable". It also equip the class with `position`, as a `THREE.Vector3`. In the constructor, it takes a meshgrid and add the meshgrid to itself. It also additionally equips members like the mass and the velocity. The `mass` is a float number. The `velocity` is a `THREE.Vector3`, which is pretty fully-fledged data structure supporting a complete set of operations, like `normalize`. In light of this, the use of `THREE.Vector3` will be super helpful to the implementation of physical parts.

Bridging between the physical and the rendering tasks is enabled by the two methods: `getPhysicalProperties` and `setPhysicalProperties`. The getter will pack `position` as well as the two additional physical member variables together in an object. This object sufficiently describes the physical property of the particle, and will be passed to the to-be-implemented algorithm [6]. After the algorithm, this physical object will be updated, and passed to the setter method of `SnowParticle`. The `SnowParticle` will then update itself with regard to the three members.

```
class SnowParticle extends THREE.Object3D {
    constructor(mesh) {
        super();
        this.add(mesh);

        this.velocity = THREE.Vector3();
        this.mass = 0.0;
    }

    getPhysicalProperties() {
        return {
            position: this.position,
            velocity: this.velocity,
            mass: this.mass
        }
    }

    setPhysicalProperties({position, velocity, mass}) {
        this.position = position;
        this.velocity = velocity;
        this.mass = mass;
    }
}
```

Listing 1: SnowParticle class

2.3 Shader

To render the snow in a realistic way, the key is to make a `THREE.Mesh` with a geometry and a shader. The geometry part is pretty straightforward. It's just a sphere with a reasonable radius (depending on the spacial density of the snow particles in the physical simulation). The shader, on the other part, is crucial in the rendering task.

There are two types of shaders applied to the snow particle mesh-grids. The (optional) vertex shader will determine the displacement of vertexes in the mesh-grid. The fragment shader will determine the color of the fragments. Some built-in shaders, like `THREE.MeshPhongMaterial`, provide a pretty decent effect for demo and tests, yet are still way from realistic for snow rendering.

In view of the necessity to have custom shaders, we figured out the mechanism to plug-in hand-written GLSL shaders into `THREE.js` framework. Shaders are handled by `THREE.ShaderMaterial`, which allow you to pass-in a "shader object" upon construction. In minimum, a shader object should have three members, the first two are code snippets of a vertex shader and a fragment shader. They are all written in GLSL and will be compiled dynamically in runtime. The third member is `uniform`, which will inject some typed global constants into the stream processors in GPU.

After a `THREE.ShaderMaterial` is instantiated, it can then be used to construct a `THREE.Mesh`, along with a `THREE.Geometry`. For snow particles, a spherical `THREE.SphereGeometry` is ideal. Listing 2 shows a naive `THREE.ShaderMaterial` that simply casts the vertex into camera coordinate space and then render it with a pure color. It is then used to construct a `THREE.Mesh` which will later be used to define the appearance of snow particles (See Section 2.2).

```
const vertexShader = `
varying vec3 vUv;

void main() {
  vUv = position;

  vec4 modelViewPosition = modelViewMatrix * vec4(
      position, 1.0);
  gl_Position = projectionMatrix * modelViewPosition;
}`;

const fragmentShader = `
uniform vec3 color;
varying vec3 vUv;

void main() {
  gl_FragColor = vec4(color, 1.0);
}
`;

const uniforms = {
  color: {type: "vec3", value: new Color(0xffffff)}
}
const shader = new THREE.ShaderMaterial({
  uniforms: uniforms,
  vertexShader: vertexShader,
  fragmentShader: fragmentShader
})

const geometry = new THREE.SphereGeometry( 0.06, 32, 32 )
    ;
```

```
30 const sphereMesh = new THREE.Mesh( geometry, shader );
```

Listing 2: Use custom GLSL shaders

2.4 Front End User Interface Design

Since it is a web based application so we want to add some interactions for users to interact with our simulation including changing the size of particles, changing the number of particles.

Based on *Javascript* and *HTML*, we also designed a user interface for our WebGL application. We just designed a simple interaction interface right now and we will add more features as we finish the physical properties this week so we can know how many hyperparameters we can change to view the difference among rendering results. Figure 3 shows one basic UI element to change the size of the particle and the number of particles in the scene.

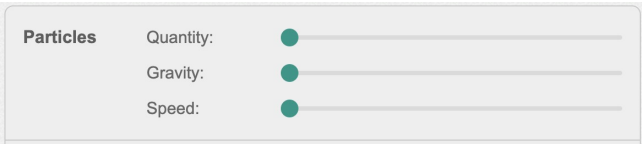

Figure 3: UI Component to control the particle properties.

3 FUTURE WORK

Based on our established codebase, we will continue working on the parallel parts in the next week.

Physical Characteristics. The first part is to implement the physical simulation algorithm [6] in JavaScript. Luckily, we have plenty of materials as reference. Besides the original paper, we may also refer to existing C++ implementation like [2] and [3].

Shader. We will need to figure out the way to render the snow particles in a realistic style and implement it in GLSL shaders. A move we will probably to take is e-mailing the authors of [6] as the original paper already made an amazing rendering in its figures. We will also try to learn from some fancy WebGL-based open-source projects like [1] which has many sophisticated shaders included.

REFERENCES

[1] [n. d.]. edankwan/The-Spirit. https://github.com/edankwan/The-Spirit. Accessed: 2019-04-30.
[2] [n. d.]. gitAzmisov/snow. https://github.com/Azmisov/snow. Accessed: 2019-04-30.
[3] [n. d.]. JAGJ10/Snow. https://github.com/JAGJ10/Snow. Accessed: 2019-04-30.
[4] Jos Dirksen. 2013. *Learning Three. js: the JavaScript 3D library for WebGL*. Packt Publishing Ltd.
[5] Tony Parisi. 2012. *WebGL: up and running*. " O'Reilly Media, Inc.".
[6] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. 2013. A material point method for snow simulation. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 102.
[7] Jian Tan and Xiangtao Fan. 2011. Particle system based snow simulating in real time. *Procedia Environmental Sciences* 10 (2011), 1244–1249.