Assignment #4 – Simple particle effect: Snowing

Yoke Kai Wen, 2020280598

April 21, 2021

1 Introduction

In this assignment, I created an animation of snowflake particles with different sizes falling with different constant speeds, starting with less snowflakes and gradually increasing the number over time. Figure 1 shows screenshot of the snowing effect.

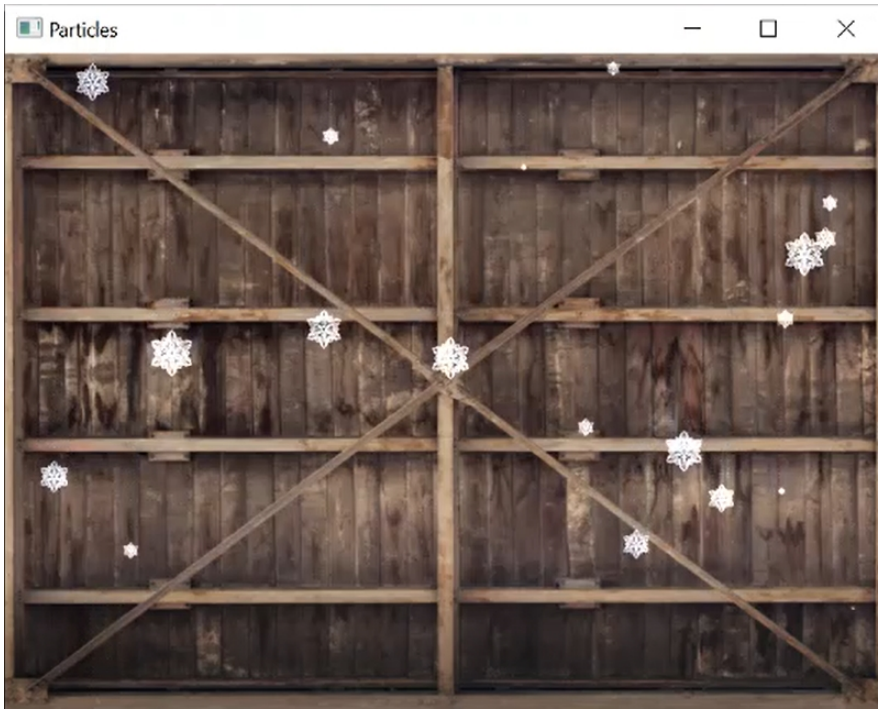

Figure 1: Screenshot of snowing animation

2 Files and directory structure

1. Main script: main.cpp
2. Header file for snowflake particles: snowParticles.h (in include/tut_headers/ folder)
3. Shader files in shaders/: main.vert.glsl, main.frag.glsl, bg.vert.glsl, bg.frag.glsl
4. Texture files in textures/: snow3.png, container.jpg

5. Other header files: include/tut_headers/shader.h

6. Set include/ and lib/ folders as include and library directories respectively in VS.

3 Usage

In `main.cpp`, the user can set parameters to determine the maximum number of snow particles which controls the duration of snowing, the average speed of snow falling, and the density of snow.

```
BunchofSnow* snowflakes;
int numParticles = 100;
glm::vec3 speed_init = glm::vec3(0.0, 60, 0.0);
double density = 2.0; //density of snow, time interval betw snowflakes is 1/
    density

snowflakes = new BunchofSnow(numParticles, (int)WIDTH, (int)HEIGHT, speed_init,
    density);
```

4 Design

4.1 Snowflake particle attributes

Each snowflake particle is defined by a `Snowflake` class with the main attributes `size, position, color, speed`. Each particle is initialised with a random size, random position starting from the top of the window, transparent colour, and a random speed as seen in the code fragment below.

```
//Snowflake class initialisation
void init()
  {
    this->size = rand() % 50 + 5; //random size
    this->position = glm::vec3(GLfloat(rand()%width),height,0);
    this->color = glm::vec4(1.0, 1.0, 1.0, 0.0); //not visible
    this->speed.y = speed.y + rand() % 20 - 10;
  }
```

4.1.1 Snowflakes of different sizes

Each snowflake particle is initialised with a constant size scale factor, which is randomly generated from a range of 5 to 55. The scale factor is then passed into the vertex shader and multiplied to the snow particle position coordinates to adjust the particle size when rendering the snowflake onto the window as seen in the code fragment below.

```
//vertex shader
void main(){
    float scale = size;
    TexCoords = vertex.zw;
    ParticleColor = color;
    gl_Position = projection * vec4((vertex.xy * scale) + offset, 0.0, 1.0);
}
```

4.2 Falling motion of snowflakes

Each snowflake falls at a constant speed from the top to the bottom of the window. This is achieved by altering the y-coordinate, reducing it from $y = HEIGHT$ to $y = 0$ with every iteration of the game loop. When the particle falls below the screen, the particle is initialised again and the falling motion restarts, as seen in the code fragment below.

```
//Snowflake class update
void update(double dt)
  {
    position -= speed * GLfloat(dt);

    if (position.y < 0) {
      init();
    }
  }
```

4.3 Increasing number of snowflakes over time

There is a set maximum number of snowflake particles `snowNum`. At every instant, all `snowNum` particles are rendered, however, most of them are invisible due to the alpha parameter being set to zero. At every fixed interval, one snowflake is made to fall from the top and becomes visible. This interval is controlled by the parameter `density`, such that the time interval between two new snowflakes is $\frac{1}{density}$. When all `snowNum` snowflakes have been made visible, and all snowflakes fall below the window, it appears as if the snowstorm has stopped. See code fragment below.

```
//BunchofSnow class update function
void update(double dt)
  {
    for (int i = 0; i < snowNum; ++i)
      snowflakes[i]->update(dt); //update position

    numSec += dt;
    if (numSec > 1/density && numVis < snowNum) {
      snowflakes[numVis]->color.a = 1.0;
      snowflakes[numVis]->position.y = this->h;
      numSec = 0.0;
      numVis++;
    }
  }
```

4.4 Rendering snowflake texture onto quad

The snowflake is not a quadrilateral and hence some of the background in the snowflake image will show up on the quad, and this does not look good. Hence, I need to render the snowflake onto the quad without the background in the snowflake image showing. To achieve this, I first download a `snow3.png` image which has the alpha channel for the background set to zero. Then, I load and bind the image texture, and in the fragment shader, I discard fragments that have alpha below a threshold so that only non-transparent portions of the snowflake will be rendered, as seen in the code fragment below.

```
void main(){
    vec4 texColor = (texture(sprite, TexCoords) * ParticleColor);
    if(texColor.a < 0.1)
        discard;
```

```
5       color = texColor;
6   }
```

4.5 Rendering background texture image

I used another set of vertex and fragment shaders `bg.vert.glsl`, `bg.frag.glsl` for processing the background texture, for which I arbitrarily passed an image of a wooden surface `container.jpg`. In every iteration of the game loop, the background is first rendered, followed by the snow particles. In fact, I wanted to choose a nicer background, but the `SOIL` library used for image loading does not seem to work for most of the images I downloaded online, so I decided to use `container.jpg` from Tutorial 6_1.