A Snowy Day in the Mountains

Jesper Persson

December 2018

1 Introduction

This is a report on a project made for a course in advanced game programming at Linköping University during the autumn of 2018. The initial project description was to implement a particle system that runs on the GPU. Furthermore, the particles should be able to collide with objects in the scene and change direction based on the collision surface and the angle of incidence. For visualization purposes, the particles were decided to represent snowflakes.

An optional part of the project specification was an implementation of volumetrically rendered clouds. Clouds would fit the scene nicely as the snow could originate from them instead of just popping up in the sky. However, due to time constraints, volumetrically rendered clouds was never added to the project.

A fully functioning particle system running on the GPU was implemented. The particle system is updated each frame in a fragment shader by using textures as input data and output data.

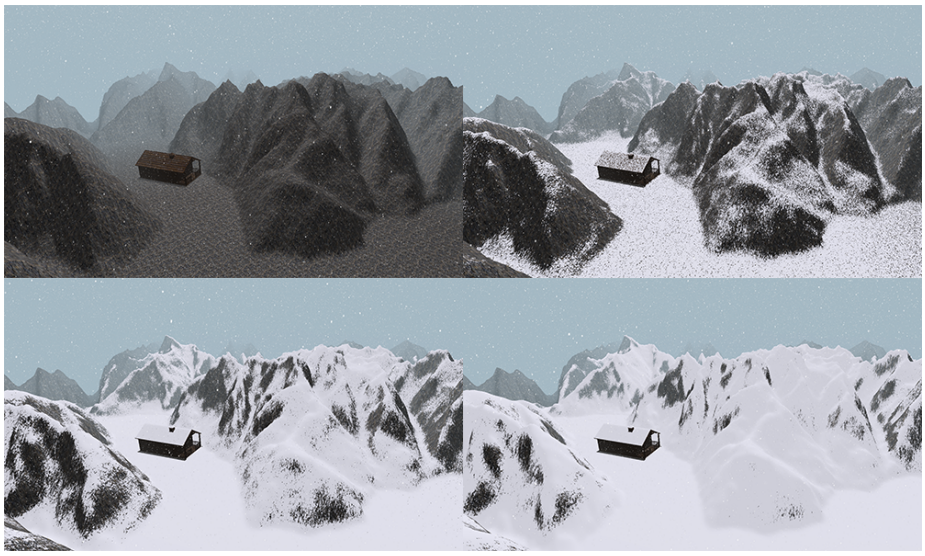

Figure 1: Scene after 0, 4, 8 and 12 minutes of snowing.

While initial collision detection and collision resolution for the particles was implemented, the project changed direction during the last weeks to focus on snow accumulation. This caused collision resolution to be left out of the final version of the code as the snow instead lands on the ground and accumulates.

The end result is a mountain scene, initially with no snow on the terrain and on the objects in the scene. As 400 000 snowflakes start falling down the sky at the same time, the terrain starts to accumulate snow on the exact location the snowflakes land on. Furthermore, the amount of snow accumulated depends on the slope of the surface the snow lands on. The result is shown in Figure 1.

2 Background

This section explains the theory used in the project.

2.1 Frame Buffer Object

Frame Buffer Objects (FBOs) [1] are OpenGL Objects that can be created by the main program. By using FBOs it is possible to render to a different framebuffer than the default one.

One or many texture objects can be attached to an FBO. When an FBO is bound, the output of the fragment shader is written into the FBO. It is also possible to output to any of the attached texture objects from the fragment shader.

When binding a FBO one specifies a *target*, which can take any of the following values: *GL_FRAMEBUFFER*, *GL_READ_FRAMEBUFFER* or *GL_DRAW_FRAMEBUFFER*. By using the last two targets, two different FBOs can be bound at the same time; one for reading and one for rendering.

After rendering to an FBO it is possible to read back the pixel data to the main memory in order to do processing on the CPU. This can be done with the glReadPixels function.

2.2 Shadow mapping

Shadow mapping is a technique that enables objects to cast shadows on themselves and other objects in the scene [2]. The algorithm consists of two steps. First, the scene is rendered from a light's position into an FBO. For each pixel, the Z value closest to the light is stored. The resulting texture is called the *shadow map*. In the second step of the algorithm, the scene is rendered from the normal camera. In the fragment shader, each fragment is transformed to the light's coordinate space, commonly known as *light space*. By comparing the fragment's Z value in light space with the value stores in the texture, a decision whether the fragment is in shadow or not can be made. More precisely, if the fragment's Z value is greater than the value in the texture, the fragment is in shadow and should thus be rendered darker.

2.3 Particle systems on GPUs

Simulating particles on the GPU enables vastly more particles than the same simulation on a CPU. In an article on Gamasutra [3], a state-preserving particle system running on the GPU is introduced. The idea is to represent particle data, such as position and velocity, using the pixel values of textures. The particle system is updated by invoking a fragment shader for each texture pixel. The output data is used as the input in the next simulation step.

When rendering the particles, the vertex shader in invoked for each particle and its vertices. The shader reads from the position texture and uses the pixel value for the current particle to translate the vertices.

To enable collision detection in one direction, the same technique as shadow mapping uses can be employed. Assuming that the particles fall from above, colliding with convex meshes is simple to implement. First, the Z value of fragments when rendering the scene from above is put into an FBO. Secondly, the normal scene is rendered. Each particle is converted into the same coordinate space as when rendering the scene from above. Just like in shadow mapping, the Z values are compared to detect if a collision has occurred or not.

A collision response can be calculated by finding the partial derivatives of the depth texture from the first render pass. The symmetric derivative formula [4] can be used on all non-edge pixels. Calculating the symmetric derivative in X, Y and Z yields the slope of the surface.

2.4 Multi-pass shaders for image blurring

Blurring images can be achieved by using a multi-pass shader. One FBO is used as input to the shader, while another FBO is being written to. After each shader pass the roles of the two FBOs are interchanged. This is referred to as ping-ponging [5].

To achieve a blurring effect, a convolution kernel can be used. A convolution kernel for Gaussian blur is shown in Table 1. The numbers of the matrix represent how much weight each pixel gets when calculating the new value for the central pixel. The total weight should sum up to 1.

0.0625	0.125	0.0625
0.125	0.25	0.125
0.0625	0.125	0.0625

Table 1: Convolution kernel for Gaussian blur.

During each shader pass all pixels are convoluted with the kernel. The neighboring pixels to each pixel are multiplied with the corresponding kernel value. This becomes the new value for the central pixel. This is performed for each pixel over multiple shader passes.

2.5 Shader Storage Buffer Objects

Shader Storage Buffer Object (SSBO) [6] is a Buffer Object introduced in OpenGL version 4.3. It can be used to write and read data from shaders. Thus, it enables communication among different shader programs.

Just like other buffers in OpenGL, an SSBO is allocated with glGenBuffers. Next, the SSAO buffer must be bound to the GL_SHADER_STORAGE_BUFFER target, using glBindBufferBase. The second argument to glBindBufferBase is a binding point index, which is needed to access the SSBO from other parts of the program. The SSAO can be accessed in a shader the following way:

```
layout (std430, binding=2) buffer someBufferName
{
    float someName;
};
```

where binding=2 specifice the binding point index.

An advantage of SSBOs compared to other Opengl buffers is their size. The specification guarantees SSBOs to be able to store at least 128 MB.

2.6 Snow accumulation

Rendering and simulating the accumulation of snow has been investigated in several articles. H. Haglund et al. [7] uses a 2d array to store how much snow has fallen on different places in the scene. The 2d array is triangulated into a mesh and the vertices are displaced depending on how much snow they have received. The authors use a particle system to simulate the snowflakes and update values in the 2d array depending on where particles land. The particles are simple billboards representing the snowflakes.

While efficient for real time simulation, the approach by H. Haglund et al. only handles static scenes in which no objects move. This limitation is handled in a paper from 2015 by D. T. Reynolds et al. [8], where each object in the scene is mapped to its own height data array, instead of one grid for the entire scene. This allows objects to move around the scene, while having their snow cover following them around. The authors use a depth texture rendered from above to find areas that are not occluded, and thus can receive snow. This depth texture must be recreated when objects move, which is not needed in the approach by H. Haglund et al.

3 Implementation

This section goes through how each feature was implemented.

3.1 Language and libraries

The main program is written in C++ and shader programs are written in GLSL. The program uses GLFW to get an OpenGL context and a window. GLEW is used to load pointers to the OpenGL functions. LodePNG is used to load PNG data into memory. Tinyobjloader is used to load obj files. GLM is used for vector and matrix mathematics.

3.2 General

The terrain in the scene is created using a heightmap image. It is rendered using a phong shader program. Shadows are produced using shadow mapping.

3.3 Particle system

The particle system builds on the principles introduced in [3]. When the program starts, two textures with four 32 bits components per pixel are generated on the CPU. Each pixel in the textures corresponds to a particle. The first texture holds the positions while the second texture holds the velocities.

Since a position in 3d only needs three float values, both the position texture and velocity texture has one component free. The forth component of the position texture is used to indicate how much time left in seconds the particle has to live. The forth component of the velocity texture is used to avoid having a particle register collisions multiple times.

The two textures are attached to an FBO. This way the fragment shader can write to the textures. After each update, two different texture objects are bound to the FBO and the previous once are used as input instead. This is needed because a shader cannot read and write to the same texture.

Besides the position and velocity texture there are two textures containing the initial positions and initial velocities of the particles. This results in four textures being bound.

The fragment shader is invoked for each pixel (which corresponds to a particle). By using the interpolated texture coordinate each particle finds its current position and velocity in the textures.

During the fall, air resistance and gravity are the main forces acting on a snowflake. Since air resistance is proportional to the velocity squared, it will eventually cancel out the gravity causing a constant speed for the particles after a while. Next, the net force is used to find the new velocity according to:

$$V_{new} = V_{old} + F_{net} \cdot dt \tag{1}$$

The position is updated according to:

$$P_{new} = P_{old} + V_{new} \cdot dt \tag{2}$$

In the above two equations, *dt* refer to the time elapsed since the last frame.

It was investigated to use a 3d texture to represent the force distribution over the scene, as a way to simulate wind. The result did not look as good as envisioned and the force field was left out from the final solution.

The particles are rendered using instance rendering. Each particle is a billboard. By using the *gl_InstanceID* variable the correct world position of a particle is found form the position texture. This is used to translate the vertices.

3.3.1 Collision detection

The collision detection mechanism makes use of a depth texture as explained in the background section. First, the scene is rendered from above and the Z value of each fragment is stored in a texture. From here on this texture is referred to as the *depth texture*.

When updating the particle system, the depth texture is available in the shader. Each particle is transformed to the same coordinate space as the depth texture was produced in. Next, the z value of a particle is compared to the color value of the depth texture. When a particle collides, its velocity is set to zero, and the forth component of the velocity texture is set to 1, indicating a collision has occurred.

During the initial development, collisions resulted in the particles sliding down in the direction of the gradient to the surface. This was achieved by calculating the derivative in X, Y and Z on the depth texture. As snow typically sticks when it lands, and because of the focus on snow accumulation, this part was left out from the final solution.

3.4 Snow accumulation

The snow accumulation in this project builds on top of the method presented by Haglund et al. [7]. The depth texture, which is created by rendering the scene from above, is used to build a new mesh that cover the entire scene. The depth texture is thus interpreted as a heightmap. But all pixels in the depth texture should not be connected, as different pixels belong to different objects in the scene. Therefore, three vertices is only connected to a triangle if the vertices of the potential triangle are at about the same height. If a triangle is not created, the vertices are considered laying on the edge of an object. Therefore, they should not be displaced in the Y direction as the snow accumulates. If they were displaced, the snow cover would not have any sides, just a floating top. The forth value of the position is used to indicate an edge. From this point on, this mesh will be referred to as the *snow mesh*.

The normals of the snow mesh are not calculated from the depth texture. This is because the depth texture has a much higher resolution than the heightmap used for the terrain. Using the depth texture to calculate normals

would introduce visible flat squares in the rendered image since the normals would be the same for many neighboring triangles.

Instead, the normal map is calculated on the GPU by taking the depth texture as input. Next, the normal map is run through a low pass filter about 20 times to get rid of sharp edges. The low pass filter uses a Guassian convolution kernel, as shown in the background section. The resulting normal map can be seen in Figure 2. Figure 3 shows the difference of the snow when using a blurred normal map.

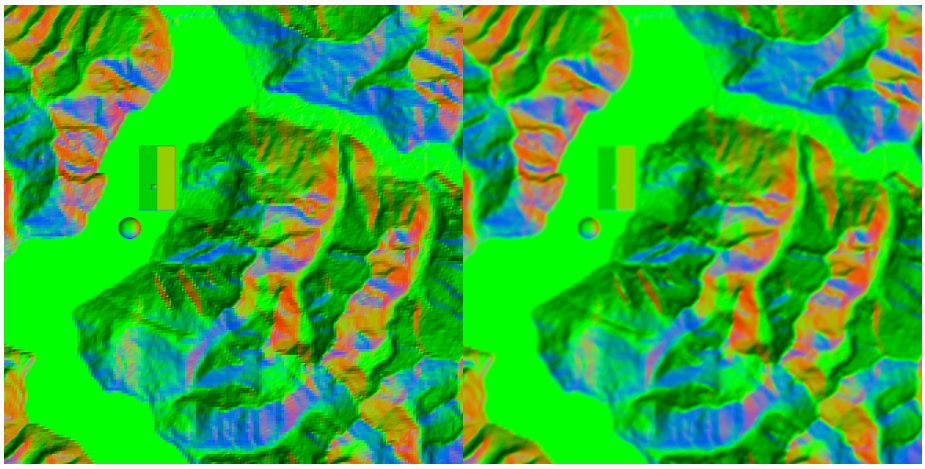

Figure 2: Left image shows pixelated normal map. Right image shows smooth normal map.

Figure 3: Left image shows pixelated terrain. Right image shows smooth terrain.

When a particle collides with the ground, the X and Y coordinates of the particle are used to index a floating point value of an SSBO, previously allocated by the CPU. Each X and Y coordinate map to an specific index. The value

at the given index of the SSBO is increased. The amount of increase when a snowflake collides depends on the normal of the surface, and the angle of incidence, according to:

$$increase = max(0, dot(-normal, normalize(velocity.xyz)))$$

where *normal* is the surface normal and *velocity* is the particle's velocity. This enables different snow accumulation depending on the direction of the snow, as shown in Figure 4 where the snow falls at an angle.

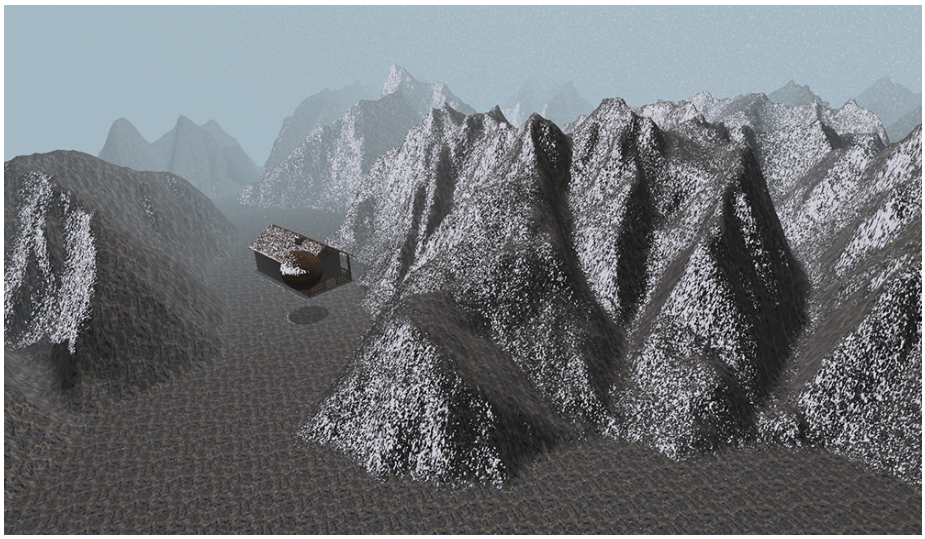

Figure 4: Snow accumulating on the side of the mountain due to side wind.

When rendering the snow mesh, the vertex shader uses the input vertex coordinate to fetch data from the SSBO at the correct location. Next, the Y value of the vertex is increased depending on the value in the buffer, as to simulate the accumulation of snow. Figure 5 shows how the snow on the roof of a cottage is accumulated.

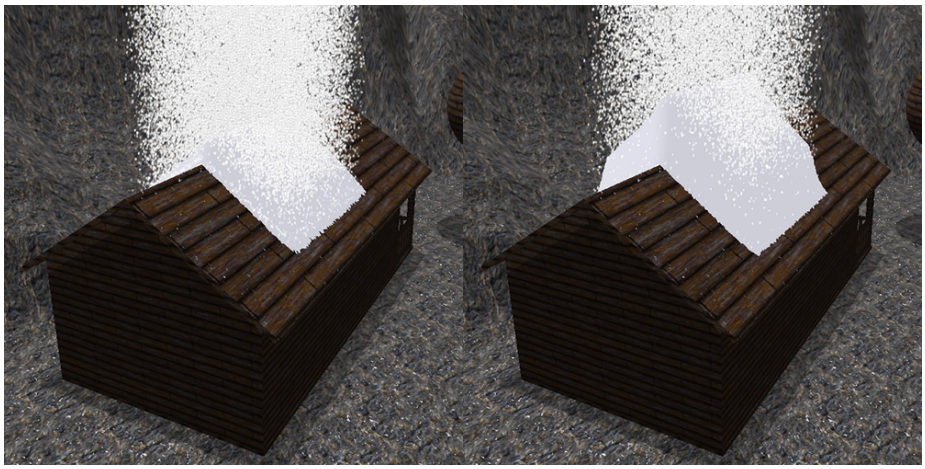

Figure 5: Snow accumulating on cottage.

Due to limited resolution of the snow mesh, making an entire triangle white when a snowflake collides would cause bad looking snow. In real life, a snowflake does not cause a solid white patch when it lands. Especially not in the shape of a triangle. While it looks good at a distance, it does not look good close-up. To get detailed and varying snow shapes when viewed close-up, noise textures are used.

When rendering the snow mesh, the value in the SSAO that maps to a certain fragment is used as a threshold to decide if a fragment should be discarded or not. The threshold is compared to the pixel value of the noise texture, according to:

```
if (noiseTexture.r >= numParticleCollisions) {
    discard;
} else {
    outColor = vec4(0.95,0.95,1,1) * lighting;
}
```

As *numParticleCollisions* approaches 1, the full triangle is drawn. But at the same time, the neighbouring triangles have accumulated snow as well, which avoids the look of triangular patterns.

Three different noise textures, created in Photoshop, are used to avoid repeating patterns. Figure 6 shows one of the noise textures.

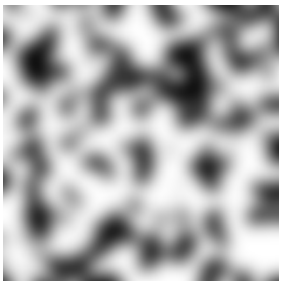

Figure 6: Noise texture used to get a nice shape of the snow on the terrain.

To decide which of these three noise textures a given triangle should use, a forth noise texture is used according to:

```
int i = mod(int(outerNoiseTexture.r * 100), numNoiseTextures);
noiseTexture = noiseTextures[i];
```

4 Discussion

The implementation section discussed several problems that were solved throughout the project. These include pixelated terrain due to low resolution heightmap relative to the depth texture and poor looking snow close-up. This section focuses on the problems in the project that were not addressed.

4.1 Collision detection

The initial vision for the project was to have snow fly in all directions, due to wind, and collide against arbitrary meshes. However, as shown in Figure 7, using a depth texture to detect collisions limits the complexity of the scene. When the snow collides with the top-most box, it starts to slide down in the gradient's direction. As the snow slides over the edge, it falls down to the bottom-most box, and start to follow that box's gradient. Next, the snow jumps up to the top-most box again, since the patch under the top-most box is not represented in the depth texture.

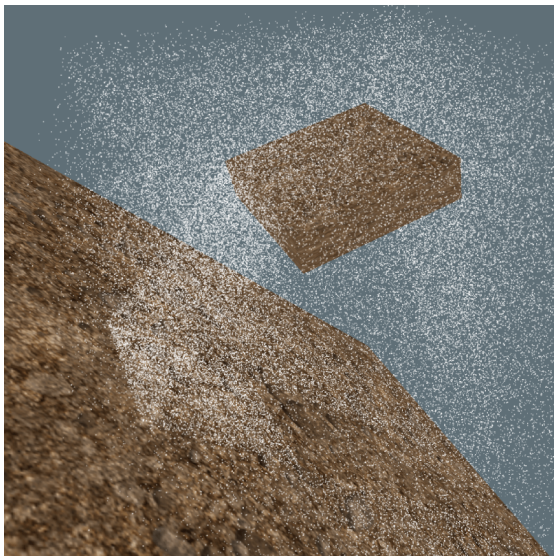

Figure 7: Limitation of depth texture collision detection.

In a static scene, using multiple depth textures from different locations and angles could be used to mitigate this problem.

A section issue with particle collision, due to the limited resolution of the depth texture, is that the different heights a particle can land on are not continues. This can be seen in Figure 8. In shadow mapping, this is referred to as shadow acne, and can be solved by adding a offset to the depth texture comparison [9]. However, the same solution is not possible here. Instead, linear interpolation when sampling the depth texture could be used; however, this would also interpolate pixels that represent two entirely different heights and objects.

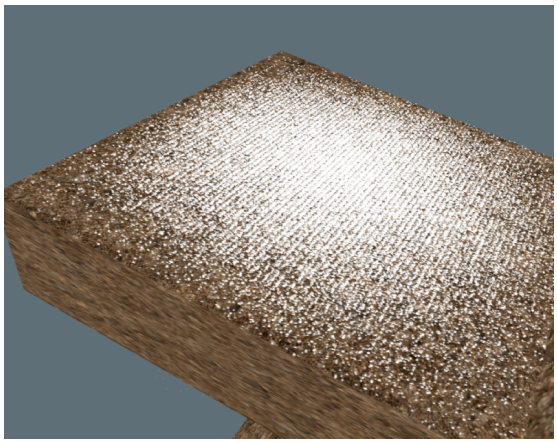

Figure 8: Particles landing at discrete Y values due to limited resolution of heightmap.

4.2 Snow accumulation

The snow mesh comes with some limitations. Since the snow mesh is created from discrete height values, it can intersect the terrain at times. A solution to this is to move it up slightly in the Y direction. How much it needs to be moved up depends on the resolution of the depth texture and how fast the height of the terrain varies.

A second issue with the mesh creation is jagged edges of objects not aligned with the axis of the depth texture. This can easily be seen on a sphere, since a sphere cannot possibly have all triangles be aligned with the texture axis. Figure 9 shows the issue. Therefore, the snow mesh should not be generated this way. Instead, individual objects in the depth texture should be identified and get their own mesh. This could potentially be implemented in a shader that takes the depth texture as input, and outputs a texture where the pixels have a value of 1 if their neighbors have a similar value in the depth texture, 0 otherwise. Next, this texture could be read back to the CPU and a mesh could be created for each object using some polygon triangulation such as the ear clipping method [10]. Doing this would resemble the work done in [8], where different meshes in the scene get their own snow mesh. A side effect is that it enables a dynamic scene where objects can move around freely.

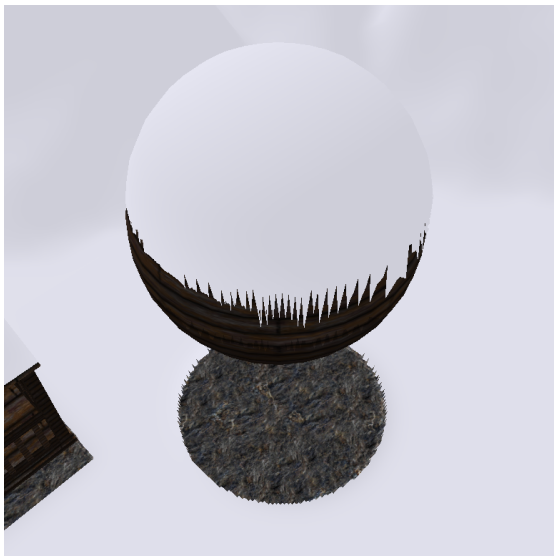

Figure 9: Jagged edges of objects that do not line up with the depth texture coordinate axis.

A third issue, also caused by limited resolution, is building a snow mesh to handle small areas. It was investigated to have the snow fall on trees. But the branches turned out to be too small for the mesh, causing poor looking snow. To handle small objects in a big scene, multiple depth textures could be used so that small objects get a higher depth texture resolution.

Another issue pertains to the lighting and look of the snow. The snow builds up differently when it comes from the side as opposed to falling straight down. But the normals are calculated with the assumption that the snow falls straight down. This causes the lighting to be wrong. Furthermore, the lighting on edges is not correctly calculated, as it does not take into account that the edge vertices are not displaced. Furthermore, bumpmapping is suggested to get more details in the snow.

A fifth issue concerns the simulation time. If the simulation runs too fast, the snow does not land evenly on the terrain. This causes some areas to get unnatural looking spikes in the snow. A solution to this might be to run the SSAO through a low pass filter to even out such spikes.

Finally, the noise textures add variation to the shape of the snow as it lands on the ground. However, the textures do not take into account how much snow has accumulated in neighboring triangles. Therefore, bad-looking edges occur where textures meet each other. Generating tileable noise textures might address this issue.

4.3 Performance evaluation

To evaluate the performance of the particle system, the snow accumulation was turned off in order to study the particle system in isolation. (Particle collision detection was still in use.) Figure 10 shows the Frames Per Second (FPS) for four different simulations. Each measurement of the FPS is the average of 10 consecutive FPS indications from the program. The measurements show that doubling the number of particles roughly cuts the FPS in half.

The setup used for the evaluated had an Intel i5 2500k CPU and an NVIDIA GeForce GTX 660 GPU.

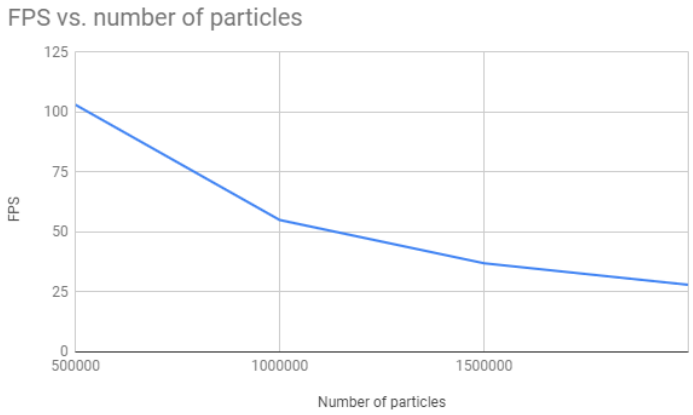

Figure 10: FPS for different number of particles.

A. Kolb et al. [11] received 10 FPS at 1 000 000 particles using a similar system to simulate particles and perform collision detection. That is significantly less that the result of this project, however they used much older hardware.

In a master thesis from 2016 [12] the author simulated 2 284 544 particles on a NVIDIA GeForce GTX 760 with an update time of 0.00826 seconds and a render time of 0.01657 seconds. This corresponds to 40 FPS. While that project had a different aim and implementation, the result is still comparable.

The collision detection and collision resolution in this project were implemented in the same shader. Separating these tasks could potentially improve the performance. The problem of having a shader doing both collision detection and collision resolution is that not all particles will collide. This means that the control flow will be different among the threads on the GPU. Due to the Single Instruction Multiple Data (SIMD) nature of GPUs, having different threads take different paths in the program causes slow-downs [13].

5 Conclusions

In this project, a snowy mountain scene was created. To simulate the snow, a particle system with collision detection using a depth texture was implemented on the GPU. The snow accumulates on the terrain, using a SSBO to store how much snow each patch has received. The values in the SSBO are used to displace the Y coordinate of a snow mesh. Noise textures are used to add close-up detail to the snow.

Multiple issues has been addressed in the discussion section. Many of them pertain to the creation of the snow mesh. Besides addressing those issues, further development could focus on allowing a snow mesh to be converted back to particles. This would allow snow to fall of branches on a tree, due to a wind or too much weight from the snow.

References

[1] Khronos Group, "Framebuffer object," 2018.

[2] I. Ragnemall, *So How Can We Make Them Scream.* Createspace Independent Publishing Platform, 2017.

[3] L. Latta, "Building a million-particle system," 2004.

[4] Wikipedia Contributors, "Symmetric derivative," 2018.

[5] I. Ragnemall, *So How Can We Make Them Scream.* Createspace Independent Publishing Platform, 2017.

[6] Khronos Group, "Shader storage buffer object," 2018.

[7] H. Haglund, M. Andersson, and A. Hast, "Snow accumulation in real-time," *Proceedings from SIGRAD 2002*, p. 11, 2002.

[8] D. T. Reynolds, S. D. Laycock, and A. M. Day, "Real-time accumulation of occlusion-based snow," *VISUAL COMPUTER*, vol. 31, no. 5, pp. 689 – 700, 2015.

[9] I. Ragnemall, *So How Can We Make Them Scream.* Createspace Independent Publishing Platform, 2017.

[10] Wikipedia Contributors, "Polygon triangulation," 2018.

[11] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-based simulation and collision detection for large particle systems," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, pp. 123–131, ACM, 2004.

[12] A. Larsson, "Real-time persistent mesh painting with gpu particle systems," Master's thesis, Linköping University, 2016.

[13] I. Chakroun, M. S. Mezmaz, N. Melab, and A. Bendjoudi, "Reducing thread divergence in a gpu-accelerated branch-and-bound algorithm," *CONCURRENCY AND COMPUTATION-PRACTICE EXPERIENCE*, vol. 25, no. 8, pp. 1121 – 1136, 2013.