UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Deformable Snow Rendering, Rendering von Schneeverformungen

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Artur Wasmut

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Kevin Keul, M.Sc. (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Abstract

Accurate snow simulation is key to capture snow's iconic visuals. Intricate methods exist that attempt to grasp snow behaviour in a holistic manner. Computational complexity prevents them from reaching real-time performance. This thesis presents three techniques making use of the GPU that focus on the deformation of a snow surface in real-time. The approaches are examined by their ability to scale with an increasing number of deformation actors and their visual portrayal of snow deformation. The findings indicate that the approaches maintain real-time performance well into several hundred individual deformation actors. However, these approaches each have their individual restrictions handicapping the visual results. An experimental approach is to combine the techniques at reduced deformation actor count to benefit from the detailed, merged deformation pattern.

Zusammenfassung

Eine genaue Schneesimulation ist der Schlüssel zur Erfassung der charakteristischen Visualisierung von Schnee. Aufwendige Methoden existieren, die versuchen Schneeverhalten ganzheitlich zu ergreifen. Die Rechenkomplexität dieser Ansätze hindert sie daran, Echtzeitfähigkeit zu erreichen. Diese Arbeit stellt drei Methoden vor, die unter Verwendung der GPU eine echtzeitfähige Deformation einer Schneeoberfläche darstellen. Die Ansätze werden hinsichtlich ihrer wahrheitsgetreuen Schneedarstellung untersucht und nach ihrer Fähigkeit, mit einer zunehmenden Anzahl von schneeverformenden Objekten zu skalieren. Die Ergebnisse zeigen, dass die Methoden bei mehreren hunderten schneeverformenden Objekten ihre Echtzeitfähigkeit beibehalten. Jedoch benachteiligen die charakteristischen Einschränkungen jener Methoden die visuellen Resultate. Ein experimenteller Ansatz ist es, die Anzahl der schneeverformenden Objekte zu reduzieren und durch Zusammenfügen der Methoden ein genaueres, kombiniertes Verformungsmuster zu erzeugen.

Contents

1 Introduction

Snow is incredibly difficult to simulate truthfully in computer graphics. Its behaviour is nearly impossible to capture within a uniform framework. To that end, several methods have emerged that are flexible enough to simulate snow masses of varying properties. Recent developments lead to the creation of the material point method used in Disney's popular movie Frozen. However, these methods' huge drawback is their computational complexity and consequently their time consumption. At best case scenarios, several minutes are required to render a single frame.
This thesis highlights several methods which are able to run in real-time when given reasonable restrictions. Specifically, the explored methods pertain to the deformation of snow terrain. Due to continuous developments in consumer graphics hardware and their increasing role as a GPGPU [1], there is a benefit in leveraging their highly efficient power to offload homogeneous computational tasks in parallel. As each presented technique makes use of the GPU in one way or another, it is of interest to explore their performance advantage. Moreover, the techniques each put different properties into focus, hence their visual output is compared to another. The performance and visual result are weighed against the set restrictions to answer the question whether these techniques are a viable alternative.
The next section serves an overview of the previously mentioned material point method and a precursor by Sumner, O'Brien and Hodgins. Additionally, a case study of each snow deformation technique is presented. Based on this, a section is dedicated to the conceptualization of a system model and its implementation. Here, each technique is embedded into an interactive simulation. An evaluation discusses the performance and visual output of each technique separately. The experiments section explores the deformation technique's applicability to sand surfaces. Furthermore, two technique's resulting deformation is combined and applied to a surface. Finally, a conclusion is drawn that compares the summarized results with each other and an outlook for future research is given.

[1] General Purpose Graphics Processing Unit

2 Related Work

2.1 Physical Simulation

2.1.1 Animating Sand, Mud, and Snow

The work of Sumner, O'Brien, and Hodgins [SOH99] describes a technique which allows for deformation of various ground environments in response to a moving actor. Through a combination of displacement and compression algorithms with variable properties sand, mud, and snow ground behaviour may be simulated.
The ground material consists of a uniform rectilinear grid of appropriate resolution and is defined as a height field. During each time step, a ray per grid point is cast that checks for intersections with a rigid body. If the body is below the surface, the surface height is adjusted to the intersection point. Additionally, a flag is set indicating a change in height. In the next step, an algorithm generates a contour map for all flagged grid points, as seen in figure 1. The contour map contains the distance to the closest non-flagged grid point and is used for the displacement of the ground material. The subsequent displacement step redistributes the material that has been deformed by a rigid body. The grid point's contour map value is shifted to the closest neighbour with a lower value. This process is repeated until all values have been redistributed to the closest non-flagged grid points, thus forming an outline, referred to as a ring, around the rigid body. This can be seen in step C of figure 2. The grid points' height is increased and depicts the rigid body's displaced material. The displacement step only shifts the contour map's values to the first ring of unaffected grid points, hence the height is adjusted in an unrealistic way. A final erosion step readjusts the height, forming a realistic mound. At this stage, the erosion algorithm's parameters may be varied to influence the mound's form.

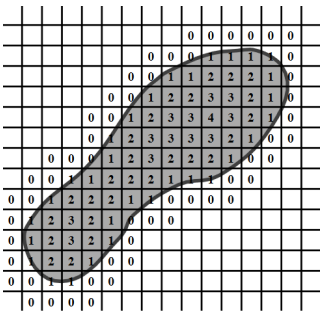


Figure 1: Calculated contour map, excerpt from [SOH99].

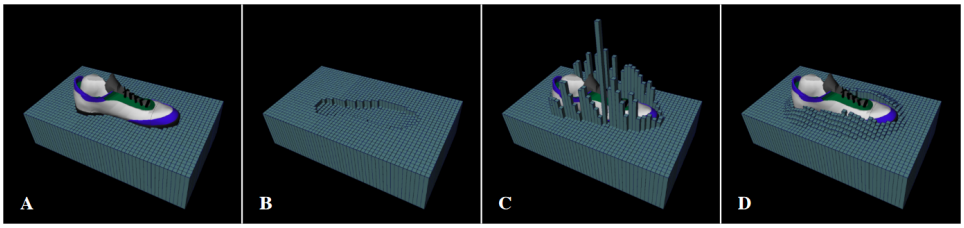


Figure 2: Individual Stages, excerpt from [SOH99].

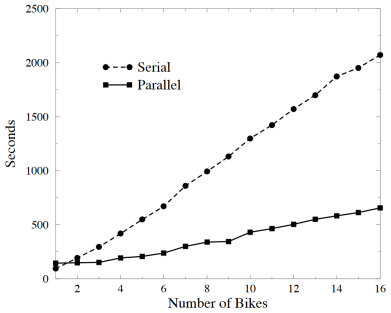


Figure 3: Timing results of one second of simulated motion per number of deformation actors (bikes). Excerpt from [SOH99].

The paper further explores options of reducing algorithmic complexity by storing the grid as a hash table and only using grid cells within the rigid body's bounding box. Moreover, a parallel implementation for multiprocessor machines is presented. Despite the improvements, considerable time is required to render a simulated second, as shown by figure 3. However, the results were captured on a Silicon Graphics Power Challenge system with 16 195MHz MIPS R10000 processors and 4GB of memory. A valid judgement is difficult to make as a comparison with modern hardware is necessary.

2.1.2 A Material Point Method for Snow Simulation

The material point method [Sto+13] is a technique using a combined approach of particle- and grid-based systems, thereby harnessing the ability to depict varied snow dynamics, covering solid- and fluid-like properties.
Based on the PIC [2] method developed by [SZS95], the idea behind MPM is to track the Lagrangian particles' data, such as mass, momentum, and deformation gradient, and propagate it to a Eulerian grid. This allows simulating mesh connectivity within the grid needed for stress-based force evaluation. In turn, the grid's calculated data is propagated to the particles, creating a feedback loop.
The procedure consists of several steps. Initially, the particles are rasterized to the grid. Using weighting functions, the particles' data is transferred. The particle's volume is estimated and passed to the grid. This volume estimation happens only once during the first time step. Consequently, the grid forces are computed. With the given data, the grid velocity is updated and grid-based body collision is resolved. The newly calculated grid data is now propagated to the particles. Here, each particle's deformation gradient as well as their velocity is updated. Finally, particle-based collision is resolved and the particle position is updated.

[2] Particle-in-Cell

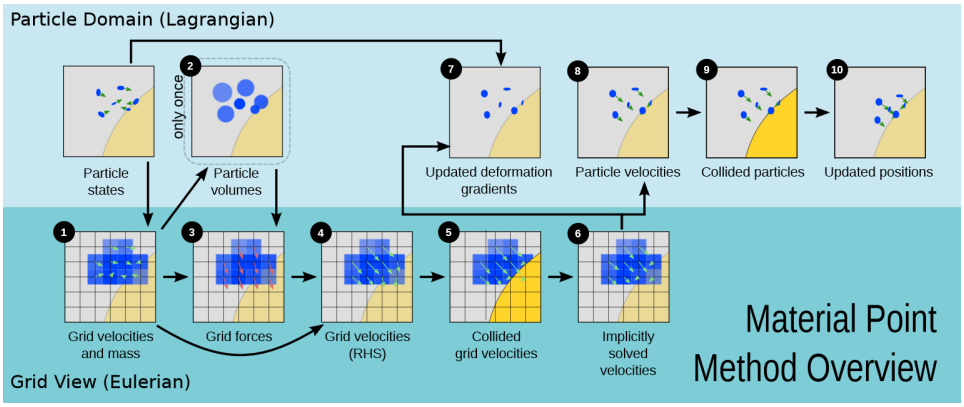


Figure 4: Material Point Overview, excerpt from [Sto+13].

The paper exploits the available parameters and presents their visual effect on the snow material. Further limitations of this method are highlighted. For example, most parameters are hand-tuned and air interaction is disregarded. Possible future research is directed at sand simulation and hardware acceleration by means of GPGPU techniques.
Although the material point method offers a visually convincing result, its main draw back is its computational complexity, therefore requiring considerable time to render a frame. Figure 5 shows the performance at different grid resolutions and particle counts for selected scenes. The results were captured on an Intel Xeon X5550 8-core machine clocked at 2.67GHz.

Example	Particles	Grid	min/frame
Snowball drop	3.0×10^5	$600 \times 300 \times 600$	5.2
Snowball smash	3.0×10^5	$200 \times 240 \times 600$	7.3
Double smash	6.0×10^5	$800 \times 300 \times 800$	13.3
Snowplow	3.9×10^6	$150 \times 50 \times 300$	2.1
Rolling snowball	7.2×10^6	$200 \times 240 \times 470$	35.7
SIGGRAPH	7.5×10^5	$780 \times 120 \times 220$	4.7
The end	5.8×10^5	$700 \times 120 \times 210$	3.8
Castle destruction	1.6×10^6	$360 \times 160 \times 560$	6.0
Walking character	3.0×10^6	$370 \times 120 \times 300$	15.7
Character digging	3.1×10^6	$280 \times 110 \times 340$	25.8

Figure 5: Simulation times for example scenes with different particle counts and grid resolutions. Excerpt from [Sto+13].

2.2 Case Studies

2.2.1 Deformable Snow Rendering

An idea to depict deformable snow is through the use of (virtual) displacement heightmaps. As explained by Barré-Brisebois in [Bar14], heightmaps for accessible rooftops and streets covered with snow are dynamically allocated based on size, characters, and visibility. To fill the heightmap with data, the snow-affecting objects are rendered in white from a bottom-up view of the surface using an ankle-high orthogonal frustum similar to figure 6.

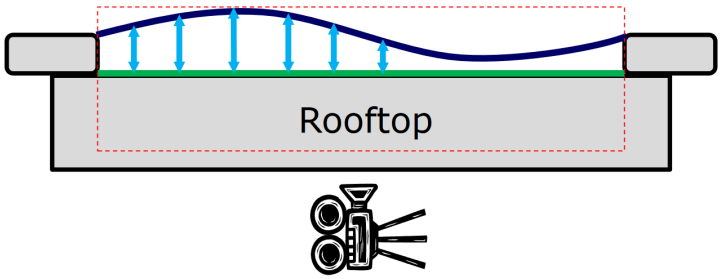


Figure 6: Schematic of the ankle-high frustum setup. Excerpt from [Bar14].

In a consecutive step, the rendered texture is post-processed; an image blur is applied. The resulting data is accumulated in a ping-pong-like fashion and added to the existing pre-allocated heightmap. Since the game is set in a snowing environment, a small value is subtracted every iteration to emulate the snow trails refilling.

Figure 7: Render target generated from the character's movements. Excerpt from [Bar14].

An assumption made here is that the snow is placed on flat geometry, therefore virtual displacement is achieved via relief mapping and requires no raycasts. Furthermore, the displacement is independent of the geometry's triangle density, thus an implementation on hardware such as the Xbox 360 [Mic] without tessellation features is possible. Nonetheless, an improved approach using hardware tessellation is available on PC. Here, the heightmap is treated as a depth buffer and consists of two channels, a minimum height field and the projected displacement. The displacement is applied globally to the snow surface and calculated using the height field and the normal map. Combined with an additional macro normal map, the displacement forms snow banks.

There are several advantages to using tessellated geometry. Shadows follow the surface and shift with the deformation, and the trails cast a shadow themselves. Additionally, ambient occlusion will properly occlude the trails. Performance-wise, the heightmaps update requires less than 1ms on consoles and has a low memory footprint of 2-4 MB, depending on the floating point precision.

Figure 8: Visual result of tessellated snow deformation. Excerpt from [Bar14].

2.2.2 Deferred Deformation

A modern technique presented by Michels and Sikachev in [MS16] to deform snow is called deferred deformation.
Given a terrain mesh and its corresponding snow mesh along with several dynamic objects to interact with the snow, a common approach to depict snow deformation is to render the terrain and the snow mesh into a heightmap from a top-down or bottom-up orthographic view. Afterwards, the dynamic objects' height is rendered into a deformation map using the previously acquired heightmaps to clamp the values. Finally, this deformation map is used to offset the snow mesh's height. The disadvantage is that height values from the terrain and snow need to be gathered each time and the dynamic objects require their own draw call as well.
Michels and Sikachev suggest that the necessary snow height is provided by the vertices of the snow mesh itself during the render pass, hence the heightmaps need not be rendered. Accordingly, the deformation height can be clamped when it is sampled instead of when it is rendered, allowing to pre-calculate an approximate deformation map requiring only the dynamic objects. During the rendering of the snow, the precise deformation is calculated, thus the process is deferred.

The central observation during the development of this algorithm is that the desired deformation shape is approximated by a quadratic curve. Consequently, it can be calculated using the following formula:

$$deformation\ height = point\ height + (distance\ to\ point)^2 \cdot artist's\ scale$$

Instead of rendering the objects, deformation points are used to calculate the quadratic curve. These deformation points are stored in a global buffer and a compute shader is dispatched, calculating the deformation for a given pixel area per point. Since deformation heights may overlap at any given time, atomic operations are required.

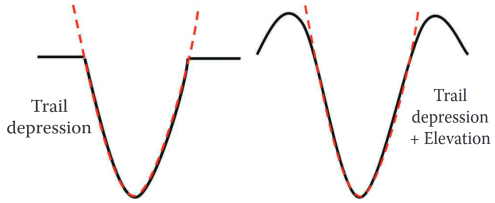


Figure 9: Trail depression with and without elevation. Excerpt from [MS16].

As seen in figure 9, trail elevation is added to the trail depression for increased overall appearance. Elevation occurs whenever the deformation height is greater than the snow height. Nevertheless, this requires additionally storing the original vertical height of the deformation point, referred to as foot height, as a trail elevation should only occur when a point is low enough to cause a trail depression. Figure 10 illustrates both possible cases where higher deformation height can occur and why encoding the foot height is necessary. To accomplish this, the 32 bits provided by the deformation texture have to be split into 16 bit each. The deformation height is kept in the most significant 16 bits for the atomic operations and the remaining 16 bits are reserved for the foot height.

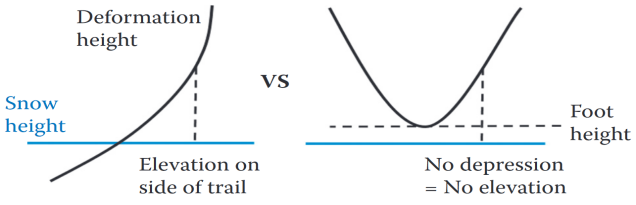


Figure 10: Schematic of deformation height above snow height. Excerpt from [MS16].

During the snow render pass, the following equations are introduced that are necessary for the elevation calculation:

$$
\begin{aligned}
depression\ distance &= \sqrt{snow\ height - foot\ height}, \\
distance\ from\ foot &= \sqrt{deformation\ height - foot\ height}, \\
distance\ from\ foot &= depression\ distance + elevation\ distance, \\
elevation\ distance &= distance\ from\ foot - depression\ distance.
\end{aligned}
$$

The depression distance is defined as the distance between center of deformation and end of depression. Similarly, the distance from foot is defined as the distance between center of deformation (the foot) and the current point being rendered. The final variable, elevation distance, is in direct relation to the two previous variables. The elevation distance is the distance between start of elevation and the current point being rendered. Figure 11 highlights the variable's relationship and shows that the deformation map's data along with the snow height provides enough information to calculate the elevation distance.

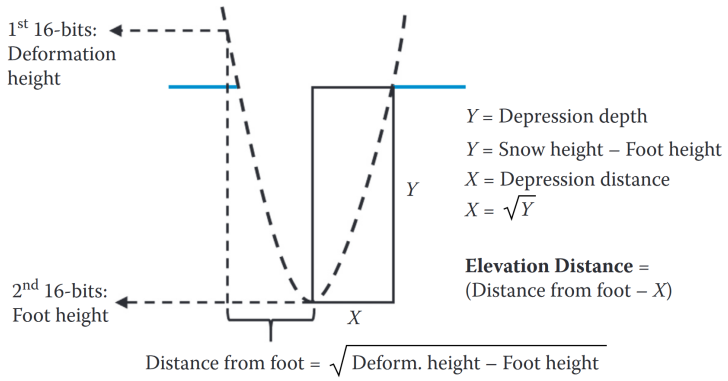


Figure 11: Relationship between deformation data and the distances. Excerpt from [MS16].

A maximum elevation distance is calculated factoring in the depression depth to scale the elevation with regard to the produced deformation. The ratio of current and maximum elevation distance is used in a quadratic function, creating the rounded elevation edges of the trail. The following is proposed in [MS16]:

$$
\begin{aligned}
ratio &= \frac{elevation\ distance}{max.\ elevation\ distance}, \\
height &= max.\ elevation\ distance\ \cdot\ artist's\ scale, \\
elevation &= ((0.5 - 2x\ ratio)^2 + 1) \cdot\ height.
\end{aligned}
$$

2.2.3 Screen Space Decals and Billboard Clouds

A different approach to simulate the impression of snow deformation is through the use of billboard or decal placements. Rather than calculating an object's actual deformation on a given surface, a simple texture or projection space is placed on the ground every time the deforming object collides with the snow mesh.

Decals A solution proposed by Kim in [Kim12] is the use of SSDs[3]. Similar to Krassnigg's deferred decal rendering technique presented two years prior in [Kra10], SSD's make use of the provided buffer data in the deferred rendering pipeline. The decal data structure consists of a 3D box mesh, called projection box, and the 2D texture decal, which can be an albedo map, a normal map or both. After the geometry pass, the projection box is rendered over the desired area and the depth buffer is used to calculate the fragments' world-space position. Using the projection box's inverse world-space matrix, the position is translated into the decal's local space. Here, the fragment is clipped depending on whether it is out of the 3D box's bounds or whether the dot product between the texture's normal and the fragment's sampled normal is greater than a certain threshold. This ensures the projected decal texture is rendered within the bounds specified by the box geometry.

Billboards [Déc+03] and [Beh+05] provide an approach to replace high poly models with so-called billboard clouds. The approach by [Déc+03] iteratively generates textures for large sets of faces by projecting them on a plane. A greedy approach chooses as many faces as possible within an error threshold to maximize the projected area and use a minimal number of planes, called dense planes. Furthermore, planes nearly tangent to the model are favoured for better surface approximation. Figure 12 shows the process applied to a helicopter model. Moreover, billboard cloud representations can be used in shadow computations. Similar to the decal approach, lighting can be improved by storing the normals in the textures.

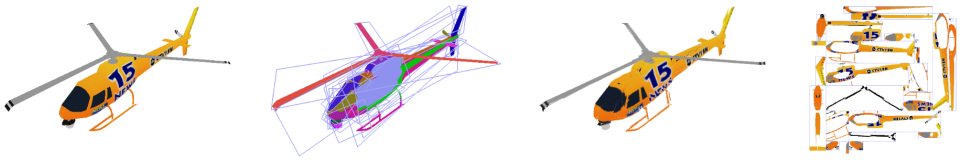


Figure 12: Decomposition of a billboard cloud (32 textures billboards) used on a high poly model (5,138 polygons). Excerpt from [Déc+03].

[3] Screen Space Decals

This idea is extended by [Beh+05] into a real-time capable landscape demo. Complex tree geometry is replaced by billboard clouds, consisting of 2-40 billboards. This allows substituting several thousand trees while still maintaining an acceptable framerate. According to the results, at 6,600 trees 9-21 FPS are achieved. At an increased 21,300 trees, the approach still manages to produce several frames per second, namely 4-12 FPS. Nonetheless, the results date back to 2005 and were generated with a 3GHz Pentium 4 processor and an Nvidia FX6800 GPU. By the same token, the resolution of 800x600 pixels is comparatively low to today's standards. A re-evaluation with updated hardware is necessary to make further conclusions.

3 System Model

3.1 Goals

Based on the previously presented techniques, an interactive simulation of snow deformation is conceptualized. The goal is to present the user a visually appealing snow landscape which is deformed by multiple spheres that bounce and slide around in the snow in a realistic fashion. A key feature of the simulation is the ability to customize deformation properties as well as switch between several snow deformation algorithms altogether. The amount of spheres is variable, allowing for a stress test of the different deformation algorithms. In this regard, interactivity specifically means that the property customization and the deformation algorithm switch is controlled by user input via mouse and keyboard, and that any changes take effect during runtime. Additionally, the scene may be reset at any given point in time, restoring the snow landscape's initial height and rearranging the spheres' position randomly, such that the simulation may be carried out again. Changes to the sphere count are reflected within the next scene reset, staying in line with the requirements set for interactivity.
In accordance with interactivity, real-time capability is another key component of the program. As the techniques are embedded in the context of animation or video games, an acceptable soft boundary to be considered real-time capable is any program that has a frame rate higher than 24 frames per second.
As for visual appeal, the goal is a combination of visual clarity, meaning the user is able to recognize the given scene as snow landscape instantly upon simulation start, and visual realism. For that matter, when comparing the program's snow deformation with reference images of actual snow deformation, the results should be as identical as possible.
The generation of additional snow geometry or snow dust via particle systems is not considered in this program. Neither are the snow's or the deforming actor's physical properties taken into account for the snow deformation.

3.2 Model Assumptions

For a concrete conception of the program, certain model assumptions have to be made. These assumptions are based on the experiences and problems encountered in the mentioned approaches in section 2.
The snow terrain is one large continuous mesh and has a fixed size that is determined beforehand. This is done to combat potential memory issues that may occur when using the terrain's dimensions to generate a heightmap. For example, the heightmaps in [Bar14] are scaled to at most one fourth of the terrain dimension and have a minimum of at least 512x512 pixels. Per frame, at most two surfaces are rendered. The main requirement of dynamic deformable snow in an open world game like Batman: Arkham Origins is a low memory footprint accompanied low performance cost. Limiting the snow terrain in this fashion accomplishes the former.
Most of the snow deformation techniques are embedded in a fully playable environment featuring human characters as the deformation actors. A fully manipulable environment featuring proper mesh and animation data for a playable character would go beyond the scope of this thesis. To that end, the dynamic actors are limited to spheres. A simple algorithm generates the mesh data for a sphere, and a small physics system calculates the movement each frame, causing the spheres to slide and bounce in the snow terrain. This creates sufficiently varied deformation while at the same time allowing the user to increase or decrease the amount of spheres without further management of the components.
Certain limitations for the physics system have to be outlined as well. The snow terrain's dimension is finite, therefore the sphere movement has to be contained within the snow terrain's boundary. Should a sphere happen to bounce or roll out of boundary, its position is corrected and the force is reverted into the opposite direction. This assures that for the remainder of the simulation run every sphere continues to stay on deformable terrain. For the sake of simplicity, the physics system does not consider collision between spheres. Although a drawback of this is that several spheres will be overlapping at some point, the physics system's complexity remains manageable and is able to scale with a higher numbers of spheres.
To create a natural-like terrain pattern, a Perlin Noise algorithm is used based on [Per02]. The noise algorithm's benefit is that the created noise data is continuous. Additionally, this is a requirement for the physics system, as it is using the height value to check for spheres colliding with the terrain. For normal calculation, the surrounding height values are required, which determine the jump or slide direction for a given sphere. Continuous height data results in a precise normal calculation, and therefore correct physics behaviour. This feeds back into the visual appeal of the simulation, since predictable and logical sphere movement creates an organic deformation trail.

A feature mentioned by Michels and Sikachev [MS16] and Barré-Brisebois [Bar14] is reaccumulating snow. The deformable snow is embedded in a snowing environment, hence having the deformed sections gradually reaccumulate back their original height further aids the typical iconic impression of snow. From a technical perspective, this serves as the deformation's soft reset. The immersion is not broken, since the environment suggests that trails should refill over time. The reset function for the simulation mentioned earlier alleviates this issue. An advantage is the ability to carefully observe the deformation caused by the actors at any given point in time, because the deformation is only reset upon user input. Moreover, the scene presented to the user is limited to a static snow terrain without any snowing. Having the trails refill over time is counter-intuitive to the presented scene.

3.3 Scene

The scene consists of a camera placed at the origin, along with a snow terrain of set dimensions with random continuous height values determined at the start. Within the boundaries of the snow terrain, an initial amount of spheres is spawned. These are placed at a certain height guaranteed to be always above the highest point of the snow terrain. Via mouse and keyboard input, the camera may be moved and the sphere physics activated. When resetting the scene, the snow terrain's heightfield is set back to its initial values. Spheres are placed at new random positions above the snow terrain, allowing the simulation to be run anew. Any change to the sphere count is now reflected. Figure 13 is a schematic representation of the scene presented to the viewer.

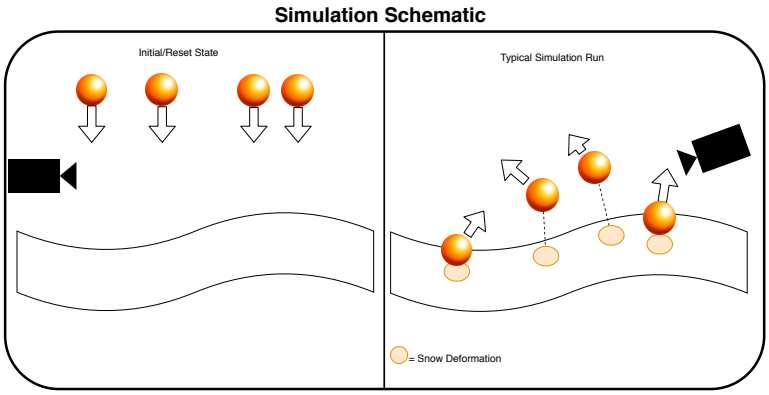


Figure 13: Scene visual schematic.

4 Implementation

4.1 Overview

At the core of this implementation is a Visual Studio C++ project using OpenGL 4.3 [Khr]. This version introduces compute shaders required for the physics component and the deferred deformation approach. A compute shader is not bound to the OpenGL rendering pipeline and allows carrying out arbitrary calculations on the graphics card. The following is a list of all external libraries integrated into the project:

- `GLFW`, utilizied for window and OpenGL context creation. This library also manages mouse and keyboard input.
- `GLEW`[4] exposes the OpenGL Core Profile.
- `GLM`[5], a header-only mathematics library.
- `stb_image.h`, a header-only image loader.
- `Dear ImGui`, a self-contained GUI[6] library.
- `Assimp`[7], to import mesh data in common formats.

Several helper functions and wrapper classes manage and abstract the interaction between user input, libraries, data and OpenGL, creating a small framework:

- a `Sphere`, `Terrain`, and a general `Mesh` and `Model` class contain or load necessary geometry data and provide a render function. A `PerlinNoise2D` subclass is contained within `Terrain` for the noise algorithm implementation.
- a `Shader` class manages shader program creation, error-logging, and compilation. It also functions as a wrapper for OpenGL's uniform variables and passes data to the specified location.
- the `Texture` class uses the image loader library to create a texture for provided image data. It can also be used for the creation of empty textures as buffer data structures.
- a `Camera` class encapsulates view-matrix calculations and receives input for movement.

[4] OpenGL Extension Wrangler Library
[5] OpenGL Mathematics
[6] Graphical User Interface
[7] Open Asset Import Library

4.1.1 Setup

A non-trivial portion of the program is executed before entering the render loop. The `Camera`, `Terrain` and `Sphere` classes are instantiated. The `Terrain` class contains the vertex data and the continuous random heightmap generated via the `PerlinNoise` class instantiated within. A `units` parameter determines the distance between neighbouring vertices. For terrain with high absolute dimensions a greater distance is chosen, so that further vertex generation can be left to the tessellation.

```
typedef struct {
  vec3 position;
  int pad1;
  vec3 velocity;
  float radius;
} Sphere;
```

Listing 1: Sphere struct with paddings for the SSBO.

```
Sphere* spheres = new Sphere[NUM_SPHERES];

for (int i = 0; i < NUM_SPHERES; i++)
{
  spheres[i].position = randomPositionWithinTerrain();
  spheres[i].velocity = vec3(0.0f);
  spheres[i].radius = 1.0f;
}

GLuint sphere_SSBO;
glGenBuffers(1, &sphere_SSBO);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, sphere_SSBO);
glBufferData(GL_SHADER_STORAGE_BUFFER, NUM_SPHERES * sizeof(
    Sphere), spheres, GL_STATIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, sphere_SSBO);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

Listing 2: Sphere SSBO creation.

The sphere movement is calculated in a compute shader, hence an SSBO[8] is created. An SSBO is a buffer structure which can be read and modified on the GPU. An advantage of SSBOs is that the storage data is available during every shader stage. A sphere struct defines the necessary position data and is copied to the SSBO. The buffer size is determined by multiplying the sphere struct's allocated memory with the number of spheres to be managed. Listing 1 and 2 show the struct definition and SSBO creation respectively.

```
// parameters consist of dimensions, formats, image data, and
    boolean flags for cubemaps or texture arrays.
Texture deformationMap(terrain_size, terrain_size , GL_RG16F,
    GL_RG, GL_FLOAT, nullptr, false, false);
glBindImageTexture(1, deformationMap.getID(), 0, GL_FALSE, 0,
    GL_READ_WRITE, GL_R16F);
glClearTexImage(deformationMap.getID(), 0, GL_RG, GL_FLOAT,
    clearValue);
```

Listing 3: Texture creation, clearing, and image unit binding.

For deformation techniques requiring a deformation map, listing 3 presents the texture creation. The textures may additionally be bound to an image unit for atomic operations within compute shaders. Since these textures constitute the deformation data that will be applied to the terrain, they are filled with a sufficiently high deformation value initially, so that any minimum operations checking for deformation will fail and leave the snow untouched.
The final part consists in setting up the shading. A PBR[9] approach is taken. The logic is based on [McA+12] and uses Epic Games' implementation within Unreal Engine 4, as presented by Karis in [KG13]. The code is rewritten to match OpenGL and GLSL[10] features and restrictions.
Specifically, several render passes are carried out to convert a provided HDR[11] map into a usable cubemap and a LUT[12] for the IBL[13] portion of the pipeline. Additionally, a random sample kernel used for SSAO[14] is pre-calculated. This concludes the shading setup as most of the PBR logic pertains to the lighting calculations made within the fragment shader.
To summarize, geometry data and camera function is instantiated; the deformation technique's data structures are created and required memory is allocated; random kernel samples for SSAO are calculated, and a cubemap with its LUT for IBL is pre-rendered.

[8] Shader Storage Buffer Object
[9] Physically Based Rendering
[10] Graphics Library Shading Language
[11] High Definition Range
[12] Lookup Table
[13] Image Based Lighting
[14] Screen Space Ambient Occlusion

The overview does not elaborate further upon procedures such as window and context creation, compatibility and error checks, frame and vertex buffer object generation, as they are not particular to the setup of the snow deformation process itself.

4.1.2 Render loop

```
while !glfwWindowShouldClose(window) do
    QueryImguiInput();
    if Deformation via Render Target then
        // Bind FBO with the deformation texture attached
           as render target
        BindFBO(rtFBO);
        SetViewport(0, 0, TERRAIN_SIZE, TERRAIN_SIZE);
        Spheres.RenderInstanced(NUM_SPHERES);
        // From top or bottom view, so it can be applied as
           heightmap
    BindFBO(0);
    SetViewport(0, 0, WIDTH, HEIGHT);
    else if Deferred Deformation then
        Deformation.Compute(NUM_SPHERES);
        // Calculate the deformation of each sphere in a
           given range
        // Written to texture that is bound as Image Unit
    else if Decals as Deformation then
        DecalManagement.Compute(NUM_SPHERES);
        // Write latest sphere position to Decal SSBO
        DecalShader.Use();
        Points.DrawInstanced(1000 * NUM_SPHERES);
        // Draw primitive geometry where decal will be
           spawned
    Terrain.RenderTessellated();
    Spheres.RenderInstanced(NUM_SPHERES);
    Physics.Compute(NUM_SPHERES);
    PollInput(&camera);
    SwapBuffers(window);
end
```

Algorithm 1: Pseudocode of the Render Loop.

Algorithm 1 presents an informal overview of the render loop. The input that `Dear Imgui` receives is queried in order to change deformation properties and switch between deformation algorithms. In the next step, the code pertaining to the selected deformation technique is carried out. Subsequently, the geometry is rendered and the terrain is tessellated. Finally, the physics calculations are done.

4.2 Snow Deformation Techniques

This section deals with the snow deformation techniques employed in the program that the user may switch between. Ultimately, three techniques are available during the program run.

4.2.1 Snow Deformation via Render Target

As the name suggests, the deformation algorithm uses a texture, referred to as the render target, where the deformation data is rendered into. In the setup step, this texture is created with a static size. For the deformation data to be written to the render target, all deformation actors have to be within the camera's view frustum at render time. For the static ground mesh the view-matrix places the camera origin at the ground mesh's center point, either facing down- or upwards. Then, the projection-matrix creates an orthogonal frustum in form of the ground mesh's bounding box. Now, the actors height is rendered and written into the texture. Since the frustum is aligned with the ground mesh's dimensions, the uv-coordinates need not be modified and can be used to sample the deformation height during the vertex shader stage. Alternatively, given the dimensions, the vertex coordinate can be normalized and used to sample with instead. This sampled value is then compared to the vertex's y-coordinate and set to the minimum of the two. Barré-Brisebois's technique [Bar14] is limited to flat terrain, allowing the use of a smaller, ankle-high camera frustum to capture the deformation actors. The render target does not explicitly contain the vertices' height value. Instead, the render target contains the deformation actors rendered in white when applying relief mapping or the resulting depth buffer is used when applying the tessellated approach. During the shading stage, the sampled value is not compared with the snow's height but used as a height offset factor in range [0;1]. Here, an intermediate post-processing step is carried out before the texture is used to be sampled from. A filter creates smooth transitions between the untouched area and the white deformation regions. This post-processing step is not available when the render target contains explicit height values. A filter falsifies absolute height values when smoothing them near untouched snow regions, creating noticeable clipping between the deformation actor and the snow surface, as the snow is not pressed down far enough. Consequently, this step is skipped.

```
glBindFramebuffer(GL_FRAMEBUFFER, rt_FBO);
glCullFace(GL_FRONT);
glViewport(0, 0, terrain_size, terrain_size);

heightSphereShader.use();
heightSphereShader.setMat4f("view", top_view);
heightSphereShader.setMat4f("projection", ortho);
sphere.renderInstanced(NUM_PARTICLES);

glCullFace(GL_BACK);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Listing 4: Code execution for render target approach.

Listing 4 shows the commands executed during the loop. Face culling is enabled to only render the geometry facing the snow.
The shader code for rendering the height is straightforward. The y-coordinate of the deformation actor is passed as an out variable to the fragment shader. The fragment shader then writes that value into the texture.

4.2.2 Deferred Deformation

This technique follows the instructions provided by Michels and Sikachev in [MS16]. They suggest decoupling the deformation from the rendering process. To that end, a compute shader is implemented that calculates the deformation within a given range of the deformation actors' location. The same SSBO storing the spheres' location and velocity for the physics compute shader is used to access the location data. The deformation map is bound as an image unit to enable read and write operations within the compute shader. Furthermore, the implementation makes use of atomic operations to guarantee that only the lowest deformation values are written into the texture at any given time in the compute step. As a consequence of that, the image format is limited. OpenGL only provides atomic operations for images of `GL_R32UI/r32ui` or `GL_R32I/r32i` format, respectively an unsigned or signed 32-bit integer precision on the red channel. Since the deferred deformation makes use of the currently calculated deformation value as well as the deformation actor's actual location for later elevation calculations, the deformation data has to be bit-shifted and packed as two 16-bit float values inside an unsigned integer. Although OpenGL provides bitwise operations and float-packing functions, this implementation utilizes Nvidia's `GL_NV_shader_atomic_fp16_vector` [NVI] extension. This allows executing atomic image operations in a convenient image format, namely `GL_RG16`, 16-bit floating point precision on the red and green channel respectively, and offers a `f16vec2` data structure where the necessary deformation data may

be put without bit-shifting and float packing.
Once the compute step finished writing the deformation depression, the snow terrain's vertex shader samples that very same f16vec2 data from the deformation texture and compares the deformation point's height to the vertex's height. A point height lower than snow height signals deformation. Here, the deformation height is compared with the snow height, determining whether the current point is responsible for snow depression or elevation. If the deformation height is lower than snow height, the current point is a snow depression point and set to the sampled value. Else, a higher deformation height signifies snow elevation. The deformation data is then used to calculate an elevation factor for smooth elevation trails along the edges of the trail depression.

```
deformationShader.use();
glDispatchCompute(NUM_SPHERES, 1, 1);
glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
```

Listing 5: Deferred deformation code execution in render loop.

Compared to the approach using render targets, the code in the render loop requires less setup and consists of a compute dispatch for each sphere as seen in listing 5. On the other hand, the compute and vertex shader have increased complexity. Listing 6 shows how each compute execution iterates over a set area around a deformation point and uses an atomic minimum to conditionally write into the image texture. Additionally, the vertex shader has several branches checking for the deformation type. Listing 7 presents an outline of the vertex shader code.

```
for(int i = -custom_range; i < custom_range; i++)
  {
  for(int j = -custom_range; j < custom_range; j++)
    {
    ivec2 deform_point = tex_coordinates + ivec2(j,i);
    float distance_to_point = (i*0.04)*(i*0.04) + (j*0.04)*(j
   *0.04);
    float16_t deform_height = foot_height + distance_to_point *
     artist_scale;
    f16vec2 deform_data = f16vec2(deform_height, foot_height);

    imageAtomicMin(deformMap, deform_point, deform_data);
    }
  }
```

Listing 6: Compute shader loop.

```
float snow_height = world_position.y;
float vert_height = snow_height;

float deformation_height = texture(deformation_map, uv).r;
float foot_height = texture(deformation_map, uv).g;

// object is stepping on snow
if(foot_height < snow_height)
{
  // depression area
  if(deformation_height < snow_height)
  {
    vert_height = deformation_height;
  }
  // elevation area
  else
  {
    float depr_distance = sqrt(snow_height - foot_height);
    float distance_from_foot = sqrt(deformation_height -
    foot_height);
    float elevation_distance = distance_from_foot -
    depr_distance;
    float max_elevation_distance = sqrt(snow_height -
    foot_height);

    float elevation = (-(pow(elevation_distance - (
    max_elevation_distance / artist_scale1), 2))) +
    depr_distance/2.0;
    vert_height = max(snow_height + elevation, snow_height);
  }
}
```

Listing 7: Deferred deformation vertex shader code.

A different elevation calculation to the one proposed by Michels and Sikachev is used, as the results were not producing rounded trail edges. The calculated elevation distance in range $[0; max]$ is offset by half the maximum elevation distance, causing it to be in range $[-\frac{max}{2}; \frac{max}{2}]$. This result is squared, resulting in a quadratic curve, similar to the depression calculations. The quadratic curve is multiplied by -1, so that the elevation increases, reaching its maximum when it is at equal distance to the maximum elevation distance and the depression distance. Once the elevation distance closes in on the maximum, it starts fading off, resulting in a smooth rounded curvature around the trail depression.

4.2.3 Billboards

A different approach to Kim's projector data structure in [Kim12] is implemented. Several assumptions can be made allowing for a deformation pattern placement without the need for a projector box. Instead of using decals, flat billboards are placed at the sphere's trail. The most recent locations where the spheres hit the ground mesh are captured and stored within an SSBO. A point draw call for each location is made. A geometry shader receives the point location and generates a quad where the texture is projected onto. The terrain's provided heightmap allows calculating the normal and its tangents at the point location on the spot along which the billboard will be oriented. The implementation consists of two shader programs as seen in listing 8 and 9. A compute shader passes the sphere's location data to the SSBO for the decal management and the decal geometry shader accesses the location data to draw the billboard.

```
typedef struct {
  vec3 position = vec3(0.0);
  int pad0;
} Billboard;

unsigned int counter = 0;
Billboard* billboards = new Billboard[1000 * NUM_SPHERES];
GLuint billboardSSBO;
glGenBuffers(1, &billboardSSBO);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, billboardSSBO);
glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(Billboard) * 1000
     * NUM_SPHERES, &billboards[0], GL_STATIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, billboardSSBO);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

Listing 8: Billboard struct and SSBO setup, similar to listing 1 and 2.

```
glActiveTexture(GL_TEXTURE1);
terrain.bindHeightMap();

billboardShader.use();
billboardShader.setInt("counter", counter);
billboardShader.setInt("terrainHeightMap", 1);
glDispatchCompute(NUM_SPHERES, 1, 1);
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
counter = (counter + 1) % 1000;
// to continuously fill the decal SSBO with the latest 1000
    positions where the sphere hit the snow mesh.
```

Listing 9: Billboard code execution in the render loop.

5 Evaluation

In this section the implementation's qualitative and quantitative properties are evaluated.
First, the deformation technique's visual output is captured and compared to another. Specifically, the produced deformation pattern's degree of detail is inspected and how close it resembles the deforming actor's shape. The data allows categorizing the techniques according to their use cases. For an overall outlook on the visual appearance a given snapshot of terrain deformed by each technique is contrasted with reference imagery of actual deformed snow.
Secondly, each deformation technique is tested with regard to real-time capability. To achieve this, the frame rate at increasing sphere count is measured. Compared to a physically based pre-computational approach, an increasing count of dynamic deformation actors allows making inferences about each technique's scaling potential. Additionally, details such as the deforming actor's vertex count or the deformable area's size are exploited, revealing their computational impact on each technique. Consequently, their use cases can be classified more distinctly and a differentiated conclusion regarding performance is reached.
The data was captured on a 64-Bit Windows 10 system using an Intel Core i7-3700k CPU with 4 cores clocked at 3.5 GHz. The GPU is an Nvidia GeForce GTX 960 with 2GB GDDR5 VRAM and 1,024 CUDA cores clocked at 1,279 MHz. A total amount of 16GB GDDR3 RAM is available. The installed Nvidia Game Ready driver is at version 430.64.

5.1 Visual Appearance

Figure 14 presents several reference images of actual deformed snow. It can be seen that the snow trails are visibly darker than the surrounding snow. This is due to the occlusion caused by the height differences between deformed and untouched snow sections. Another visual cue is that the trails are smoothly divided into brighter deformation that is facing the light source and a portion facing away from the light, being further occluded.
Figure 18, 19, and 20 each depict a snapshot of snow deformed by a deformation technique. The figures consist of three images. A view of the shaded result, a view of the scene's normals, and a view of the scene's calculated occlusion factor. Each image allows looking at a visually desirable property in detail.

Figure 14: Reference images. Excerpts from [Com19a], [Com19c], and [Com19b].

Deferred Deformation The deferred deformation algorithm is depicted by figure 18. The shaded result in image 18a depicts the shadows generated near the deformation as well as the snow that is shaded with a darker color. Identical to the reference image, the deformation pattern consists of areas facing the light source that are shaded in a brighter tone and darker areas facing away from the light source. This can be explained by image 18b. The new height values allow recalculating the snow's normals around the deformation area. Here, the blue and bright green color near the depression indicate the snow facing to or away from the light source respectively. Furthermore, the deformation consistently holds a higher occlusion factor as depicted by the darker color in image 18c.

Render Target Figure 19 shows the deformation caused by rendering the deformation actors' height values into a texture. Similar to the previous approach, image 19b shows how the new height values are used to recalculate the normal around the snow depression. Additionally, image 19c indicates a higher occlusion factor where the snow has been deformed. In the same way, image 19a depicts the shadows generated near the deformation and the existing areas facing to and away from the light source. A larger portion of the spheres is visible due to no snow elevation being calculated around the trail edges. Consequently, the height difference from the lowest deformation point to the trail edge is not as large. This difference is reflected in the resulting occlusion factor. The smaller the snow depression, the smaller the

concave deformation shape that is left. Hence, fewer samples contribute to a darker occlusion factor.

Decals/Billboards Figure 20 has contrasting results to the two previous deformation techniques. Notably, no actual deformation occurs and is therefore not reflected in any of the images. The deformation is achieved by rendering flat rectangular geometry with a generic deformation pattern and placing it on top of the snow. The normals are left untouched as seen by image 20b and the visible occlusion is limited to the close proximity of the spheres themselves, shown in image 20c. Therefore, the resulting image 20a depicts the sphere's shadows only and the trail is consistently dark and not affected by any light sources.

Complex Deformation Patterns A key limitation is the deformation actors' basic geometric shape in the form of a sphere. Figure 21 is dedicated to the deformation technique's deformation patterns. Image 21a shows that the implicit representation of the deformation as a quadratic function coincides well with the deformation actor's shape. Similarly for image 21b, the produced deformation texture is precise enough for a smooth round deformation pattern. Image 21c depicts the textures used for the billboards. The advantage of using spherical geometry as the pseudo deformation actor for billboards is that the resulting texture is rotationally invariant. Consequently, the texture's basis is a blue-tinted sphere whose edges have been blended. On top of that, the edges transition to a lighter color and have been smoothed with a rough brush for an additional noise pattern. The result is a texture showing a rounded deformation pattern wherever the spheres have bounced and a consistent deformation trail where the spheres have slid by repeatedly placing several textures in close proximity and blending them.
However, figure 22 offers a more distinguished look at the technique's generated pattern. A human model is used as the deformation actor in this instance. Specifically, the model's feet serve as the deformation geometry. At this point, the deferred deformation algorithm and the render target approach start differing. Because the depression step is decoupled and its own process, the deferred deformation algorithm calculates the same pattern for a given deformation point regardless of the deformation actor's geometry. Since the depression calculation is based on a quadratic function as its implicit representation, there is a discrepancy between the feet and the deformation pattern. The bottom view in image 22a shows that the feet clip through the deformed snow. By contrast, image 22b shows foot tracks with enough detail for each individual toe. With the render target approach the height values are explicitly only written to the texture positions where a bottom-up view of the geometry would have been visible.

In summary, both the deferred deformation algorithm as well as the render target approach create a convincing, permanent snow deformation. Shadows and ambient occlusion work and enhance both techniques' visual representation. The deferred deformation's calculated trail elevation serves as further hint, creating additional shadows and increasing the occlusion factor. Its deformation shape is limited to a quadratic curve. The render target approach excels in transforming the deformation actors' geometry as a deformation pattern on the snow. The billboards allow using an arbitrary texture as a deformation pattern that is unaffected by the lighting.

5.2 Performance

At test start, the camera is moved to one corner of the randomly generated snow terrain, facing the terrain center. The simulation is carried out and the most recent frame rate that is stable for several seconds is captured. Afterwards, the scene is reset and the camera moved to another corner of the terrain. The same procedure is carried out for a total of four times, until the frame rate has been captured once at every corner. This concludes one test run. For more stable results, the test run has been conducted three times, ensuring that the captured data is within margin of error and reproducible. Finally, the data is averaged. The scene is rendered at 1600x900 pixels (HD+).

32x32 Sphere				
Deformation Technique	Sphere Count			
	128	256	512	1,024
Deferred Deformation	91.75	91.5	87.5	77.25
Render Target	101.5	96.75	93	86.5
Decals/Billboards	99.75	90.875	76.5	60.5

Table 1: FPS of each deformation technique, rendering the deformation actor with 32x32 vertices.

Table 1 shows that each technique decreases in performance with increasing sphere count. At 32x32 vertices for the deformation actor, the render target approach holds the highest frame rate across all tested cases. Second in performance is the deferred deformation with a difference of about 9-10 FPS at 128 and 1,024 spheres and 5 FPS at a sphere count of 256 and 512. At 128 spheres, the decals reach a frame rate on par with the render target, although its performance decreases the strongest with higher sphere count, leading to the lowest frame rate reached at 60.5 FPS with 1,024 spheres tested.

128x128 Sphere				
Deformation Technique	Sphere Count			
	128	256	512	1,024
Deferred Deformation	93.5	91.25	85.25	76.25
Render Target	82.75	71.125	56.375	39.5
Decals/Billboards	100.5	92.5	76.75	62.5

Table 2: FPS of each deformation technique, rendering the deformation actor with 128x128 vertices.

Table 2 suggests that the render target approach relies on the deformation geometry. While the deferred deformation's and the decal's performance is within margin of error to the previous results, using geometry with high vertex count has an increasing performance penalty on the render target. At each test case, it scores the lowest frame rate with a new 39.5 FPS low at 1,024 spheres.

512x512 Sphere				
Deformation Technique	Sphere Count			
	32	64	128	256
Deferred Deformation	94.5	95	91.5	88.125
Render Target	55.5	37.125	23.5	13.5
Decals/Billboards	105.5	106.375	99.625	92.875

Table 3: FPS of each deformation technique, rendering the deformation actor with 512x512 vertices.

With a total of 262,144 (512^2) vertices per individual deformation actor, table 3 depicts an edge case. At about 350 spheres the render target approach reaches less than 10 FPS, which is why the overall observed sphere count has been reduced. At each incremental step, the render target approach has a performance delta greater than or equal to 10 FPS, with 55.5 FPS being the highest performance reached at the lowest sphere count and an absolute low of 13.5 FPS at 256 spheres. The remaining techniques continue to scale with sphere count only, reaching the same performance as before at 128 and 256 spheres. A lower sphere count indicates no significant performance penalty (less than 1 FPS).
So far, the previous evaluations indicate a consistent performance loss at increasing deformation actor count. The render target benefits from low vertex deformation geometry to render its deformation values. As expected, the deferred deformation algorithm calculates a deformation pattern regardless of presented geometry, therefore the vertex count is irrelevant. However, since a given range has to be specified for the deformation that is to be calculated, it is used as an additional performance metric instead.

Deferred Deformation Range Performance				
Deformation Range	Sphere Count			
	128	256	512	1,024
32^2	94.5	90	84	79.5
64^2	93.5	91.25	85.25	76.25
96^2	85	75.5	70.625	58.375
128^2	77.6	68.25	59.5	47.125

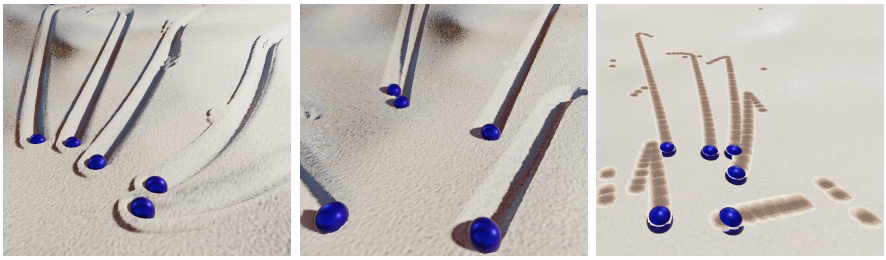
Table 4: FPS of the deferred deformation technique using various deformation ranges during the compute shader stage.

Table 4 is dedicated to the deferred deformation algorithm and its performance regarding various specified ranges. The spheres have a diameter of 2 meters, accordingly a standard deformation area of 64^2 pixels is chosen, covering approximately $6.55m^2$ at $4cm$ per pixel. Accounting for the roughly $4m^2$ of the sphere's trail depression, there is enough space left over for the trail elevation. This is the default range used in previous tables.
At a 32 by 32 pixel deformation range, approximately $1.64m^2$, no significant performance change is observed. The observations are well within margin of error. The next range increment at 96 by 96 pixels, or $14.75m^2$ signifies a greater performance delta. At 128 spheres, there is a 9 FPS loss and at 1,024 spheres the delta increases to about 18 FPS.
Analogous to table 3, the 128 by 128 pixel deformation range with a $26.21m^2$ area is its edge case equivalent. The deferred deformation algorithm reaches its lowest observed frame rate of 47.125 FPS at 1,024 spheres used. This results in a performance delta of 30 FPS compared to the default deformation range.

6 Experiments

Sand A fundamental insight about these techniques is that they are uninvolved with the rendering of the surface. For this reason, a future application mentioned both in [Bar14] and [MS16] is to reapply the deformation to various other deformable surfaces, such as mud or sand. Indeed, the method described in [SOH99] provides a proof of concept for various deformable surfaces and the material point method [Sto+13] presented in section 2 suggests further research in this direction as well.
Figure 15 presents a snapshot of a sand surface deformed by each technique. None of the techniques' implementation has been altered in any way and the terrain uses the same noise algorithm to produce the heightfield. The only change is a different set of textures used in the rendering of the deformable surface. Additionally, the billboard's projected texture is tinted with a sand-like color. A performance difference was not observed.

Edwards [Edw12] offers further methods to recreate the typical impression of sand that may be integrated into sand surfaces and combined with the deformation techniques for improved visual results.



(a) Deferred Deformation **(b)** Render Target **(c)** Billboards

Figure 15: Deformation techniques applied to a sand terrain.

Combined Techniques The evaluation's results suggest that the render target approach is suited for use cases where the deformation actor's geometry plays a significant role in the deformation process. On the other hand, for use cases where the creation of a deformation trail has higher priority and the geometry may be disregarded, deferred deformation offers visually similar results along with a trail elevation and maintains a higher performance. To that end, an experimental approach is to combine the deferred deformation's and the render target's deformation map in a complimentary manner. The desired result is to create a deformation map which allows for the calculation of trail elevations while still considering the deformation actor's geometry when forming a deformation pattern.

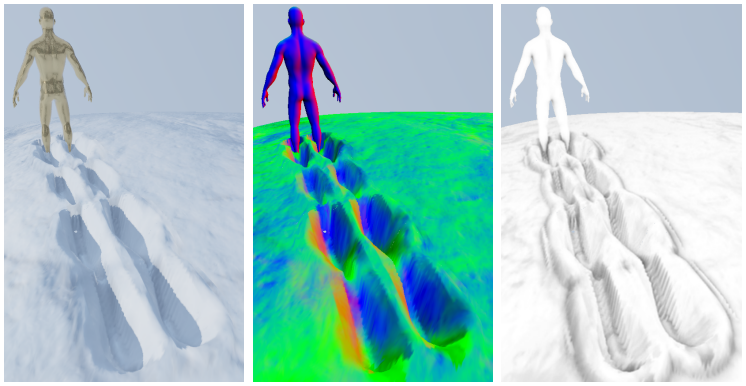


Figure 16: Deferred deformation combined with the render target approach.

Figure 16 is the result of applying the mentioned algorithms to the same deformation map in succession. The render loop now consists of setting up the camera frustum, rendering the deformation actors' height, and

dispatching a compute shader consecutively. The fragment shader unconditionally writes the height into the 16-bit red channel. In the compute shader, the SSBO's deformation point is put into the 16-bit green channel. The trail depression is calculated as is, but the atomic minimum operation now compares the texture's contained height value to the calculated deformation depression, guaranteeing that only the lower value remains. Image 17 shows how the deformation actor's foot shape remains and the region around the feet is surrounded by the calculated quadratic curve. Nevertheless, a problem occurs during the terrain's vertex shader stage. Currently, the vertex shader code has several branches that decide the type of deformation to generate. The very first branch samples the deformation point contained in the texture's green channel for further calculations, but early-outs if the value is lower than the snow height. Accordingly, expensive operations are carried out for deformed vertices only. Consequently, this means that the deformation is contained within the compute shader's set deformation range, where the green channel is written into. For deformation geometry, where the defined deformation range does not encompass its entirety, this creates regions where the geometry is below the snow surface but deformation does not occur. A solution to this issue is to change the branch conditions. Instead of checking the deformation point's height, the actual deformation height contained in the red channel is sampled. Here, the deformation is applied to any vertex whose deformation height is lower than its snow height and trail elevations are calculated for regions within the deferred deformation range. Albeit, every vertex whose deformation height is higher than the snow height additionally checks the provided deformation point's height for elevation calculations if necessary. The performance advantage of an early-out at the first branch is lost.

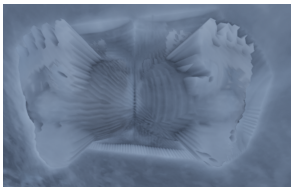


Figure 17: Combined deformation pattern.

7 Conclusion and Outlook

Three deformation techniques have been implemented in an interactive simulation. Each technique's performance was benchmarked and their visual output presented.
Deferred deformation is not dependant on geometry detail and its elevation offers an additional visual cue. However, it still does not manage to outperform the render target approach for low poly deformation actors. The deformation is limited by its approximation of a quadratic curve function. Hence, its performance depends on the defined deformation range. Possible research may be directed at the mathematical curve approximation, using polynomial functions of higher order to calculate a trail depression with several saddle points. The render targets key feature is its detailed deformation pattern allowing complex geometry to be portrayed at the cost of performance. Furthermore, the pattern is not limited to a predefined range. Consequently, large and small deformation actors alike may be rendered without having to manipulate the deformation process. Integrating both the deferred deformation and render target approach allows leveraging their individual benefits and circumventing their restrictions at the same time. However, as the spherical geometry does not benefit from the combined approach it has not been tested with regard to its performance. Therefore, a future endeavour is to test all techniques anew with varied but autonomous deformation actors that can be increased in count while maintaining similar complexity to the spheres and their physics system.
The billboard's only variable parameter is the projected deformation texture. Its main drawback is that no actual deformation is created. Nonetheless, if detailed deformation is of lesser importance, several thousand billboards can be placed per deformation actor while maintaining high deformation actor count and acceptable performance. Another advantage is that the billboards do not depend on a high poly ground mesh. It can not be ruled out that this technique has its use cases for applications with low poly geometry or restricted viewing angles not exploiting the billboard's two-dimensionality. The billboards may be extended by animated deformation textures or textures of iconographic nature. A further direction for this approach is to replace the billboards completely with Kim's projector box data structure [Kim12] and supply an additional deformation normal map. While still not causing any actual deformation, the provided normal map will influence the lighting calculations, a key feature observed in the other two techniques.

References

[Bar14] Colin Barré-Brisebois. "Deformable Snow Rendering in Batman^{TM}: Arkham Origins". In: *Game Developers Conference (GDC), San Francisco, California, March*. 2014.

[Beh+05] Stephan Behrendt et al. "Realistic real-time rendering of landscapes using billboard clouds". In: *Computer Graphics Forum*. Vol. 24. 3. Wiley Online Library. 2005, pp. 507–516.

[Com19a] Wikimedia Commons. *File: Малокозинская лыжня - panoramio (1).jpg — Wikimedia Commons, the free media repository*. [Online; accessed 20-August-2019]. 2019.

[Com19b] Wikimedia Commons. *File: Малокозинская лыжня - panoramio (10).jpg — Wikimedia Commons, the free media repository*. [Online; accessed 20-August-2019]. 2019.

[Com19c] Wikimedia Commons. *File: Малокозинская лыжня - panoramio (4).jpg — Wikimedia Commons, the free media repository*. [Online; accessed 20-August-2019]. 2019.

[Déc+03] Xavier Décoret et al. "Billboard clouds for extreme model simplification". In: *ACM Transactions on Graphics (TOG)*. Vol. 22. 3. ACM. 2003, pp. 689–696.

[Edw12] J Edwards. "Dynamic sand simulation and rendering in journey". In: *ACM SIGGRAPH*. 2012.

[KG13] Brian Karis and Epic Games. "Real shading in unreal engine 4". In: *Proc. Physically Based Shading Theory Practice* 4 (2013).

[Khr] Khronos Group. *OpenGL 4.3 API Core Profile*. [Accessed: 6-August-2019].

[Kim12] Pope Kim. "Screen Space Decals in Warhammer 40,000: Space Marine". In: *ACM SIGGRAPH 2012 Talks*. SIGGRAPH '12. Los Angeles, California: ACM, 2012, 6:1–6:1. ISBN: 978-1-4503-1683-5. DOI: 10.1145/2343045.2343053. URL: http://doi.acm.org/10.1145/2343045.2343053.

[Kra10] Jan Krassnigg. "A Deferred Decal Rendering Technique". In: *Game Engine Gems 1*. Ed. by Eric Lengyel. Jones and Bartlett, 2010, pp. 271–280.

[McA+12] Stephen McAuley et al. "Practical Physically-based Shading in Film and Game Production". In: *ACM SIGGRAPH 2012 Courses*. SIGGRAPH '12. Los Angeles, California: ACM, 2012, 10:1–10:7. ISBN: 978-1-4503-1678-1. DOI: 10.1145/2343483.2343493. URL: http://doi.acm.org/10.1145/2343483.2343493.

[Mic] Microsoft Corporation. *Xbox 360 Technical Specifications*. Archive retrieved from the original. [Accessed: 6-May-2019].

[MS16] Anton Kai Michels and Peter Sikachev. "Deferred Snow Deformation in Rise of the Tomb Raider". In: *GPU Pro 7*. AK Peters/CRC Press, 2016, pp. 17–30.

[NVI] NVIDIA Corporation. *NV_shader_atomic_fp16_vector extension*. [Accessed: 5-August-2019].

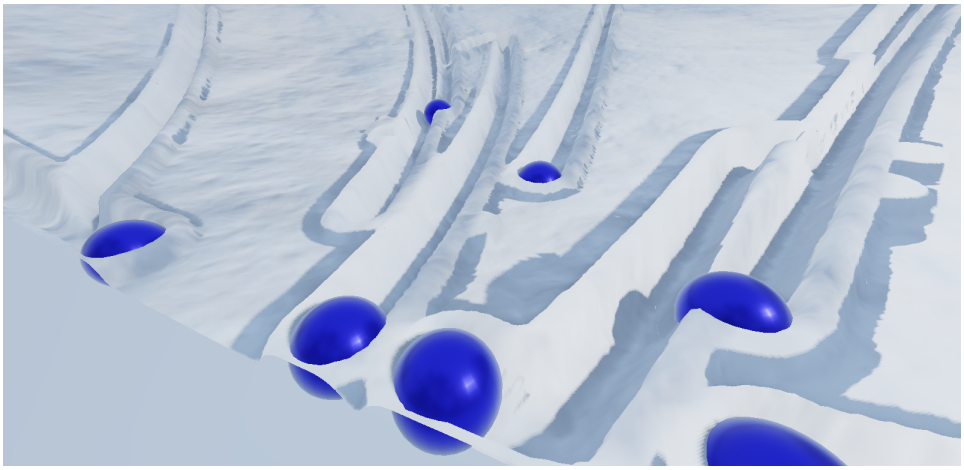
[Per02] Ken Perlin. "Improving noise". In: *ACM transactions on graphics (TOG)*. Vol. 21. 3. ACM. 2002, pp. 681–682.

[SOH99] Robert W Sumner, James F O'Brien, and Jessica K Hodgins. "Animating sand, mud, and snow". In: *Computer Graphics Forum*. Vol. 18. 1. Wiley Online Library. 1999, pp. 17–26.

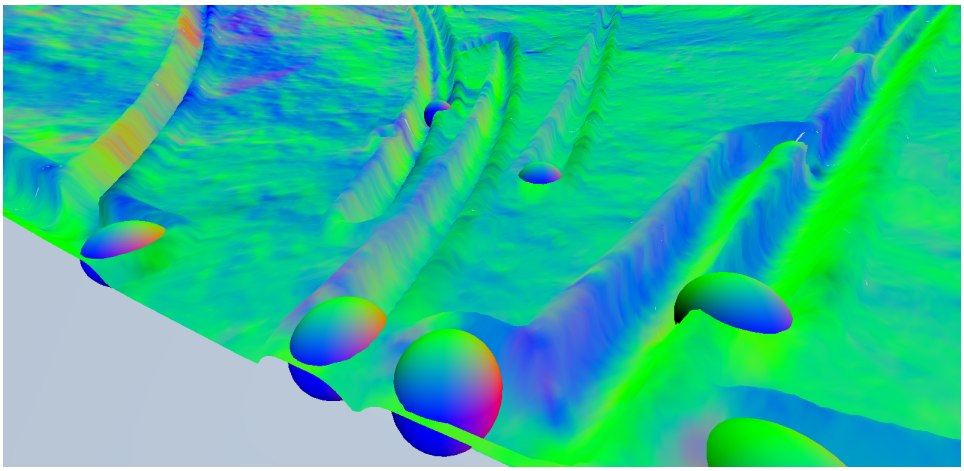
[Sto+13] Alexey Stomakhin et al. "A material point method for snow simulation". In: *ACM Transactions on Graphics (TOG)* 32.4 (2013), p. 102.

[SZS95] Deborah Sulsky, Shi-Jian Zhou, and Howard L Schreyer. "Application of a particle-in-cell method to solid mechanics". In: *Computer physics communications* 87.1-2 (1995), pp. 236–252.

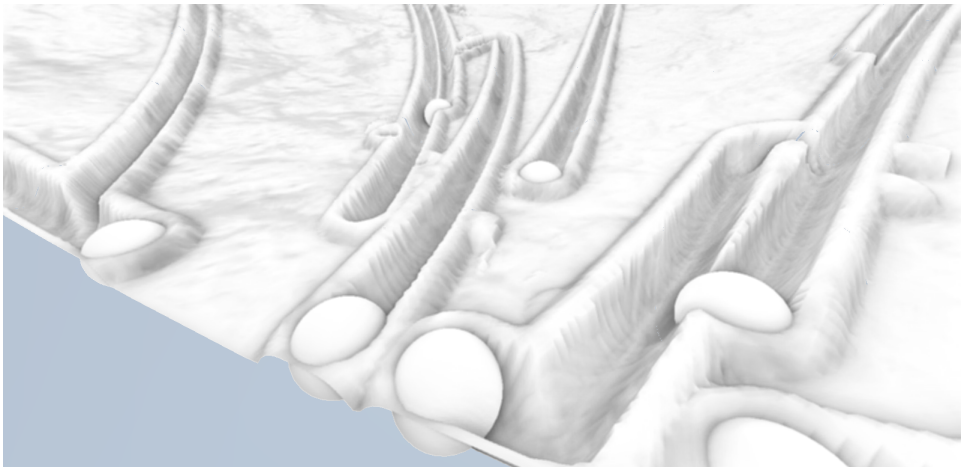
Appendices



(a) Fully shaded result.



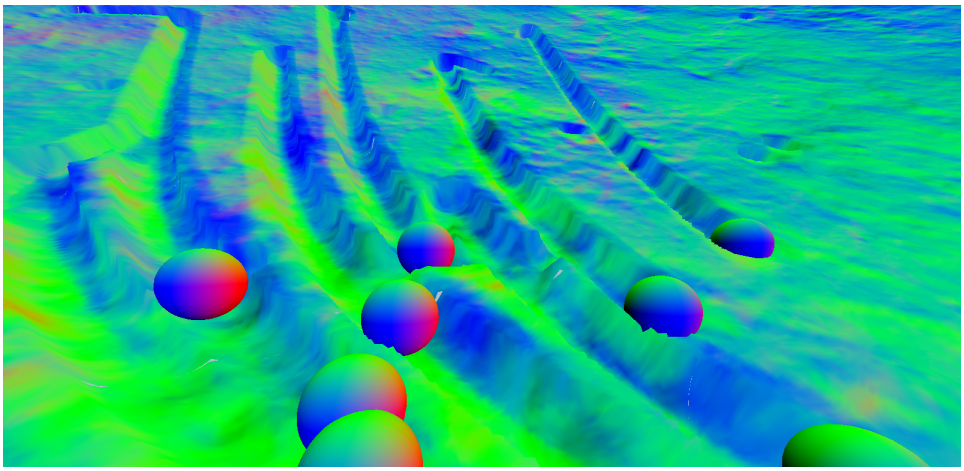
(b) Representation of normals.



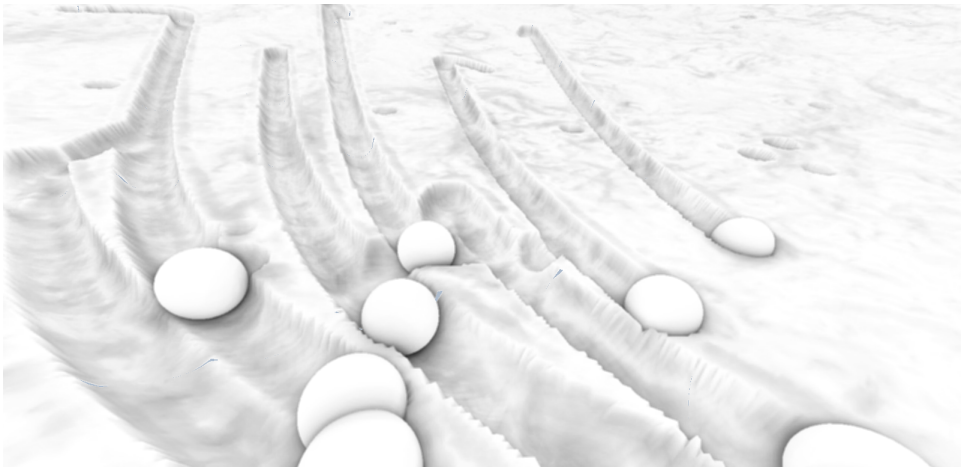
(c) Occluded areas by SSAO.

Figure 18: Snapshot of the deferred deformation algorithm.

(a) Fully shaded result.

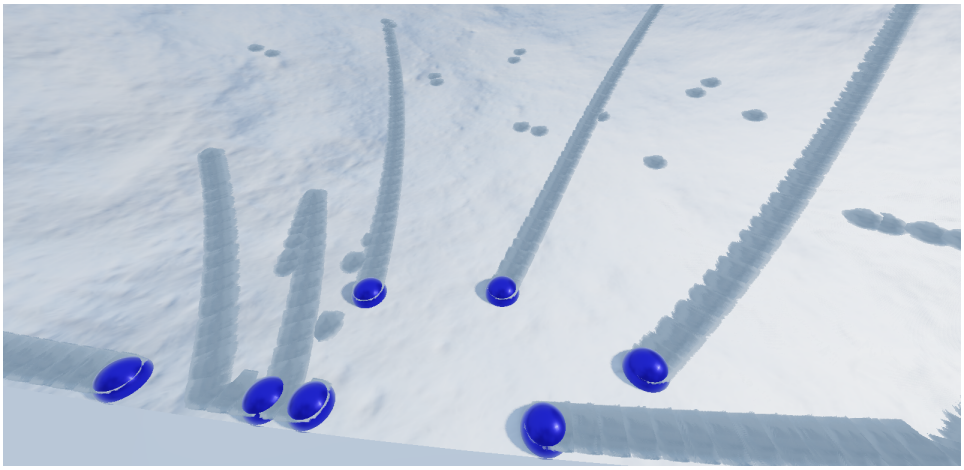


(b) Representation of normals.

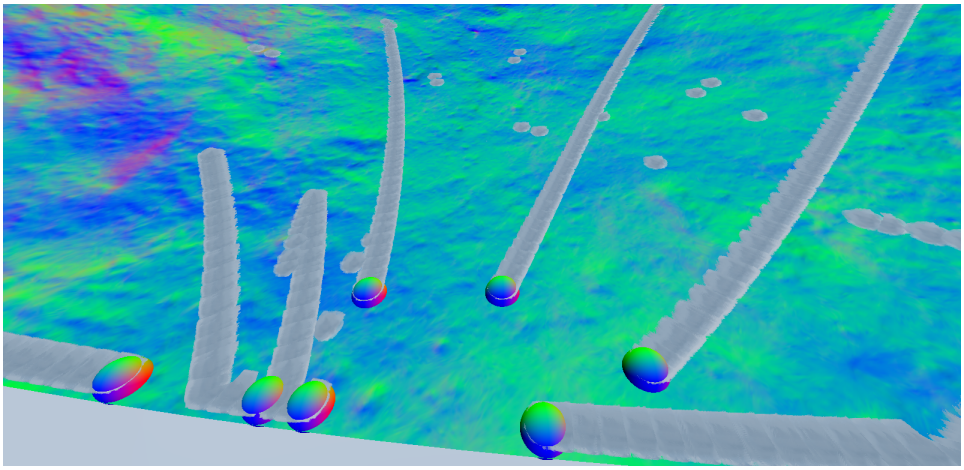


(c) Occluded areas by SSAO.

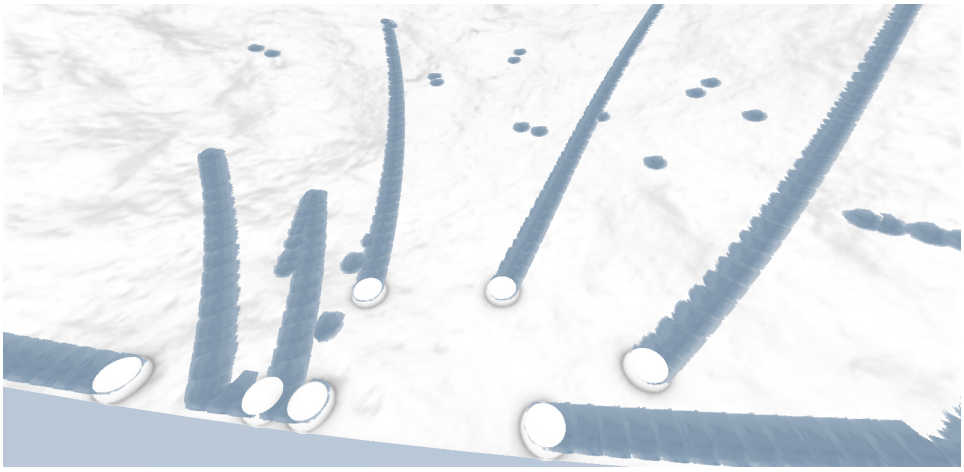
Figure 19: Snapshot of the render target approach.



(a) Fully shaded result.

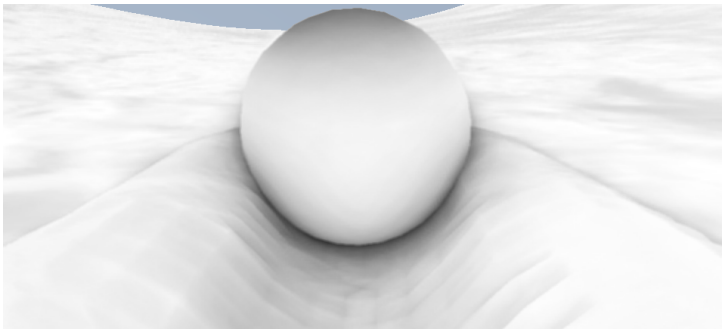


(b) Representation of normals.

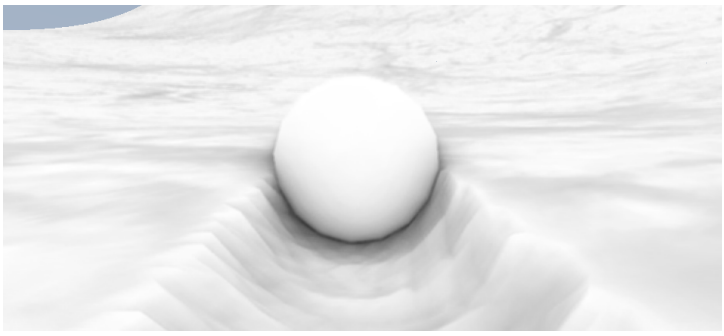


(c) Occluded areas by SSAO.

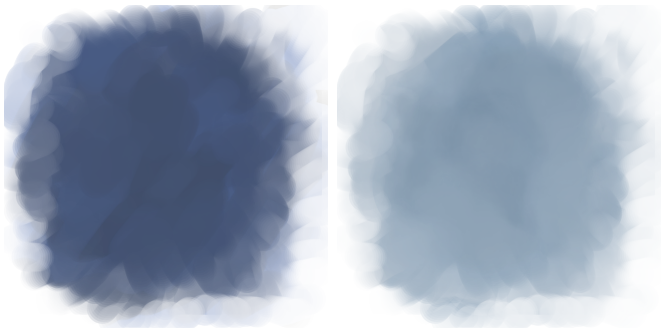
Figure 20: Snapshot of the generated decals/billboards.



(a) Deferred Deformation's deformation pattern.



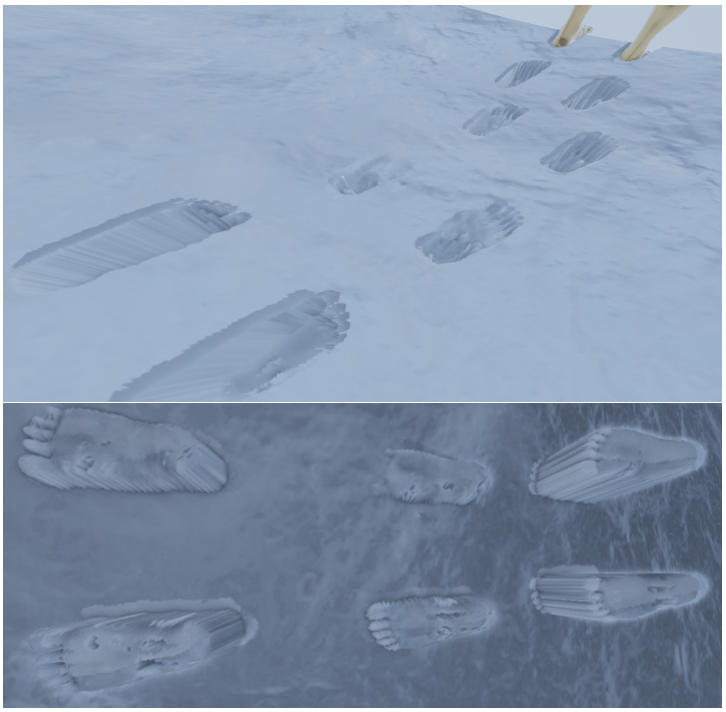
(b) Deformation pattern produced by the render target.



(c) Textures used as deformation pattern for the billboards.

Figure 21: Close up view of the deformation patterns.

(a) Deformation pattern using deferred deformation.



(b) Generated deformation using render targets.

Figure 22: Snow Tracks.