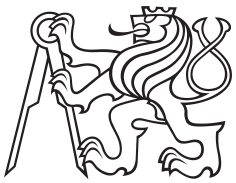
Bachelor Project



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of computer graphics and interaction

Real-time snow surface generation

Dominik Dinh

Supervisor: Ing. Jaroslav Sloup
Field of study: Open Informatics
Subfield: Computer games and graphics
January 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Dinh** Jméno: **Dominik** Osobní číslo: **491835**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Dynamický sněhový povrch

Název bakalářské práce anglicky:

Dynamic snow surface

Pokyny pro vypracování:

Seznamte se s metodou akumulace sněhu popsanou v práci Paula Fearinga [1]. Implementujte zjednodušenou variantu této metody popsanou v [2] využívající modifikovaných stínových map ke zrychlení akumulace sněhu na objektech scény. Vytvořte aplikaci pro zobrazení krajiny pokryté sněhem, jejíž povrch bude deformován pohybujícími se objekty. Na základě prostudované literatury [3-6] navrhněte a implementujte vhodnou datovou strukturu pro uložení změn tvaru povrchu způsobených interakcí s pohybujícími se objekty.
Vygenerované obrázky scén pokrytých sněhem porovnejte s fotografiemi a zhodnoťte jejich věrohodnost.
Implementaci proveďte v C/C++ s využitím OpenGL.

Seznam doporučené literatury:

[1] Paul Fearing: Computer Modelling of Fallen Snow. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp.37-46, 2000.
[2] D.T. Reynolds, S.D. Laycock, A.M. Day: Real-time Accumulation of Occlusion-based Snow. The Visual Computer, vol.31, no.5, p.689-700, 2015.
[3] A.Junker, G.Palamas: Real-time Interactive Snow Simulation using Compute Shaders in Digital Environments. In International Conference on the Foundations of Digital Games, pp. 1-4, September 2020.
[4] D.Hanák: Real-time Snow Deformation. Diplomová práce, Masarykova Univerzita, Brno, 2021.
[5] G.Cordonnier, P.Ecormier, E.Galin, J.Gain, B.Benes, M.P.Cani: Interactive Generation of Time□evolving, Snow□Covered Landscapes with Avalanches. Computer Graphics Forum, vol. 37, no. 2, p. 497-509, 2018.
[6] Benjamin Neukom, Stefan Müller Arisona, Simon Schubiger: Real-time GIS-based Snow Cover Approximation and Rendering for Large Terrains. Computers & Graphics, vol.71, p.14-22, April 2018.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jaroslav Sloup Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **01.02.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **30.09.2023**

Ing. Jaroslav Sloup podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I want to thank all the people around me, friends and family, for supporting me throughout the writing of this thesis.

Namely, my girlfriend for providing me with a shoulder I could rest my head on and for being patient with me in times of greatest stress, and my friends, especially Dan Juříček, my classmate, who had to endure my constant ranting about various topics and willingly provided me with his views on the subject. And, of course, big thanks to my family.

Two people I want to thank separately.

Doctor Daniel T. Reynolds. The author of the original method who provided me with useful insights, materials, and source code and dedicated a whole day just to help me finish the work.

Lastly, I would like to thank Ing. Jaroslav Sloup, my supervisor, for being patient and dedicating his busy schedule to help me finish my project.

Declaration

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

V Praze, 14. January 2021

Abstract

Work implementing real-time snow accumulation methods allowing for dynamic interactive scenes. Each object has it's own accumulation buffer attached in a form of texture. Main idea is using occlusion renders to determine surface exposure to snow and then using programmable geometry shader to map from occlusion renders to the unique bound accumulation buffers;

Keywords: real-time, snow, accumulation, occlusion render, geometry shader

Supervisor: Ing. Jaroslav Sloup

Abstrakt

Práce, jejíž cílem je implementovat metodu akumulace sněhu na dynamických objektech v realném čase. Každý objekt má svou akumulační texturu, ve které se uchovává výška sněhu. Hlavní novou technikou představené v této metodě je použití geometry shaderu.

Klíčová slova: realný čas, sníh, akumulace, okluze, geometry shader

Contents

Figures

Tables

ctuthesis t1606152353

Chapter 1

Introduction

Snow is a fascinating phenomenon that captivates our eyes due it's unpredictable behaviour and beauty. Creating beautiful sceneries by covering the landscape, trees, buildings with layers of cold and soft blankets. Enjoying winter time with our family or friends skiing, building snowmen and igloos and much more.

Can we recreate it on the computer? What if we wanted to create a snow covered scenery or an interactive world, like a videogame, ourselves? Inspired by the infamous game, Red dead redemption 2 by Rockstar games, I've decided to dive deeper into the world of computer generated snow.

First, we take a look at how snow behaves in the world. How snow form, why it forms and how it eventually reaches the ground. We won't go into much detail, as that's not our primary concern, but we need to understand at least the very basics, mainly, how snow forms.

Then, we are going to take a look at existing methods of snow generation. Understanding how others managed to create a working simulation will help is in better understanding the actual topic as well as aid is implementation of an existing method or even creating a brand new method.

The main focus of this paper is implementing and testing of an existing method by D. T. Reynolds, S. D. Laycock & A. M. Day [3].

Chapter 2

Snow as we know it

Before we dive deep into the secrets of computer generated snow, we have to take a brief look at how snow behaves in the real world first. We won't go into much detail as the topic itself is very complex and most information is outside of the scope of this work.

In this chapter, we lay out the requirements for snow, the different sizes and shapes of snowflakes and other important properties. This chapter is divided into two parts. In the first part, we look at how snow is created and its journey to the ground and in the second part, we look at how snow behaves when it eventually reaches its destination.

Of course, every method presented in this paper - including the one we will be implementing - is a major simplification of all the different processes and effects that happen in the real world. Many of them will even be completely omitted. However, the goal will be to give a general overview in order to understand many methods presented later this paper.

2.1 Snow in the air

This section describes the origin of snow up in the Earth's atmosphere before it reaches the ground. We will try to cover many of the conditions and factors on which the size, shape and other properties of snow depend.

2.1.1 Precipitation

Precipitation is a term used for any form of water falling down to the ground up from the atmosphere. It is one of the major components of the water cycle. There are 4 main forms of precipitation - raindrops, ice pellets, hail and snowflakes. Water vapor, in a cycle called evaporation, moves from the Earth into the overlying atmosphere, then condenses into water droplets that eventually fall down to the Earth - when they get heavy enough in a process called coalescence. Coalescence occurs when smaller droplets merge together into a bigger droplet or a droplet freezes onto an ice particle. Particles like dust and smoke (so-called "condensation nuclei"), are crucial for precipitation, as they provide enough surface area necessary for water vapor to condense upon. Given the low temperatures up in the clouds, most rain begins as snow.

2.1.2 Snow and snow crystals

We can now finally define what snow is, where it first forms, how it grows and how it's very unique path to the ground affect the its overall shape and properties.

But first, we have to clear out a common misconception. What is usually referred to as a snowflake by the general public is, in reality, a snow crystal. In other words, snow crystal is a single unique particle of ice, while snowflake is a more general term that describes, for example, a collection of snow crystals that collided together.

Formation of snow crystals

Snow crystals begin to form in altitudes where the temperature drops below 0°C as water vapor directly freezes onto a condensation nuclei. Important thing to point out is that snow crystals are not frozen raindrops - these are called sleets. The initial shape of a snow crystal is called a hexagonal prism (insert figure), which can be flat, long, thin plate-like or anything in between. It's the angle between atoms in a water molecule, that dictates the shape shall be hexagonal. A hexagonal prism consists of two basal facets ("top" and bottom") and six prism facets ("sides"). The shape depends on which facets grow faster. You get a column-like shape when the basal facets grow faster and a flat crystal when the prism facets grow faster.

Snow crystal growth

After a single snow crystal forms it continues to grow as water vapor condenses onto its surface. Corners of the prism attract more particles, hence they grow faster, giving the snow crystal it's usual hexagonal look as we know it. One large snow crystal comprises of over a million of water droplets.

First person who ever tried to fully study snow crystals was Ukichiro Nakaya, back in the 1940s. He published the infamous Nakaya diagram (see Figure 2.1) which shows how temperature and the level of supersaturation directly affect the final shape of a snow crystal.

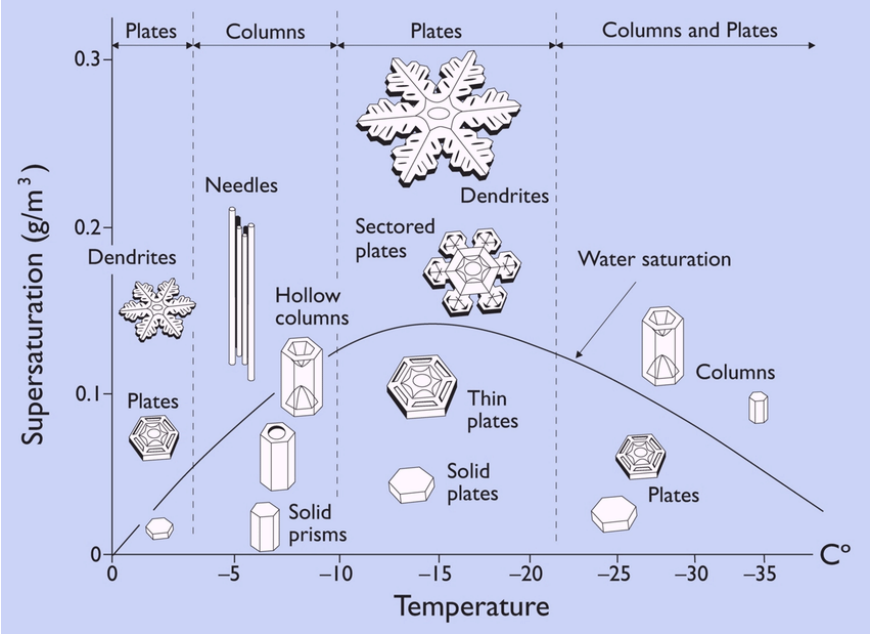


Figure 2.1: The Nakaya diagram

2.2 Snow on the ground

The snow crystal's journey ends, when it finally reaches the ground, forming snow layers along with other crystals. The goal of this section is to give an overview of how snow behaves when it finally reaches the ground and how we can interact with it.

Snowpack consists of numerous individual snow layers each made of flakes with same or similar properties. Snow changes immediately after it hits the

ground, deforming under various influences, like temperature, affecting it's overall properties.

2.2.1 Angle of repose

One of the most important aspects of snow stability. It's an angle beyond which a particular snow type is very likely to move, causing the snow to change positions and shape. It also dictates what surfaces can hold snow.

2.2.2 Avalanches

An avalanche is a very fast flow of snow down a slope, such as a hill or mountain. Avalanches can be set off spontaneously, by such factors as increased precipitation or snow pack weakening, or by external means such as humans, animals, and earthquakes.

Avalanches occur in two general forms, or their combinations, slab avalanches made of tightly packed snow, triggered by a collapse of an underlying weak snow layer and loose snow avalanches made of looser snow. After being set off, avalanches usually accelerate rapidly and grow in mass and volume as they capture more snow. If an avalanche moves fast enough, some of the snow may mix with the air, forming a powder snow avalanche.

Chapter 3

Snow simulation methods

We now know how snow behaves in the real world. But how do we transform this knowledge into an effective and working simulation method? As with everything in computer science, the goal is to take a complex problem and simplify it as much as possible while preserving the original idea. How to achieve it?

There are numerous already existing methods that try to simulate snow as realistically as possible. Of course, every situation is different and each method is useful in a different scenario. Identifying the requirements in each scenario is key for choosing or creating a correct method resulting in a believable representation.

Some methods aim to provide a real time solution. This might be useful in a dynamic environment, such as games, where the world changes constantly. These methods are, most often than not, much less realistic than their static counterparts - computation power is a big limitation requiring a very simplified solution that allows us to simulate the snow in multiple frames per second.

Methods, that don't care about real time use cases, are more realistic because we don't need to limit ourselves to just a few milliseconds - duration of a frame. We can take advantage of much more complex algorithms and techniques that may require minutes, or hours to fully complete. The usual use case would be generation of a static scene with little to none moving objects.

Actual techniques used vary greatly and it's difficult to grasp all of them. However, many of them take advantage of the same or similar algorithms or techniques, but use them differently - from particles to GPU tessellation or compute shaders.

Overall, there are two main approaches to snow modelling, physically and non-physically based. In this chapter, we introduce and compare different techniques used to simulate snow.

3.1 Non-physical approaches

Most, if not all, non-physical approaches are based on occlusion taking advantage of techniques like shadow mapping, displacement mapping and more.

3.1.1 Occlusion based methods

As in the real world, snow should not settle on a surface, that is not visible from the direction of snow fall. In other words, if a surface is blocked by a different object, it should not accumulate snow. The main idea behind an occlusion based snow simulation is similar to shadow mapping. First, we need to render the scene from the perspective of the source - an arbitrary position from which snow falls. The result is a depth map which we can use to determine if a given surface is visible from the direction of snow fall and hence receive snow.

Ohlsson and Seipel

One of the methods utilizing occlusion was introduced by Ohlsson and Seipel [[18]].

Snow coverage is calculated using accumulation prediction function split into two parts. Utilization of occlusion happens in the second part, the exposure function, which calculates how occluded the surface is. Everything is done realtime, allowing for some degree of interactivity, but due to snow

accumulation being recalculated every frame, should an object suddenly move into occlusion, all of the accumulated snow would be lost instantly.

Performance wise, with a screen resolution of 600 x 600, in a scene consisting of 16000 triangles the average frame rate achieved was around 13 frames per second, making this method unacceptable in a very dynamic environment like games.

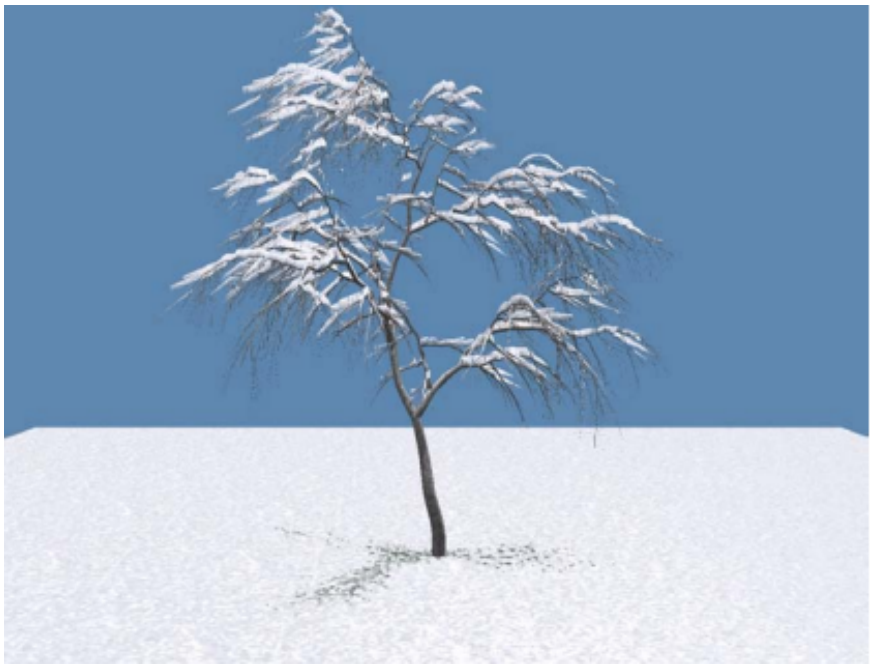


Figure 3.1: The result of applying described method to a more complex object.

Related work

Other noteworthy occlusion based methods include [[19]], [[5]]. Method proposed by Tokoi is also real-time, unfortunately giving even worse fps performace.

3.2 Physical approaches

Physically based approaches aim to simulate snow in a more realistic manner, focusing more on the geometry utilizing particle systems, height maps and more.

3.2.1 Paul Fearing

Method that inspired many future works is a method proposed by Paul Fearing in his dissertation [[4]]. Particles are shot towards the sky from so-called launch sites in a snowflake simulating pattern - slightly varying its direction to imitate flake flutter - and traced to determine the exposure to falling snow.

Fearing was also the first to use polygonal meshes. To achieve greater level of detail, mesh was dynamically subdivided using the Delaunay triangulation, resulting in Veronoi diagram-like mesh subdivision, with each face having its own particle launch site, allowing for a greater level of detail as each face contains information about the accumulated snow.

This method overall is quite complex and not too easy to implement. The computation takes a long time and the simulation needs to run to the very end, consequently no build-up animation is possible. The result is completely static scene with no interaction allowed. The method is also unsuitable for real-time uses.

To illustrate this method's complexity, simple scene containing a single object, fire hydrant (see Figure 3.2), took about 15 minutes to finish.

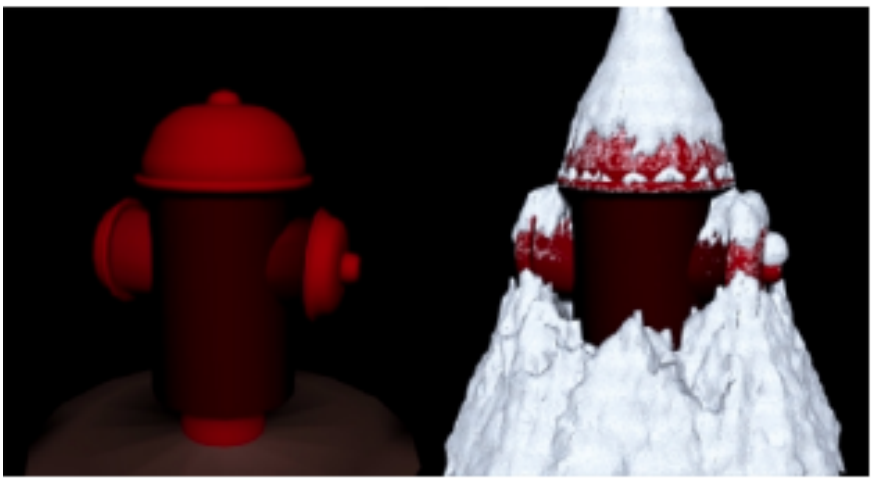


Figure 3.2: Fire hyndrant with accumulated snow taken from Fearing's paper[4]

3.2.2 Others

A very interesting physically based method also takes advantage of particles, however, unlike Fearing's method of using them to trace the path to the sky, the snow is actually comprised of numerous particles creating an interesting effect [[16]]

Chapter 4

Real-time accumulation of occlusion-based snow

The main point of interest leading to this work was snow in video games. In other words, I was looking for a method, that would allow us to simulate realistic looking snow in an ever-changing environment, with an added bonus of interactivity.

However, snow accumulation and interaction are two very complex topics on their own, which makes it very difficult to focus on both of them at the same time, while preserving the real-time requirements along with the visual requirements. Best looking methods usually don't work in real-time and methods with the best interaction aren't very visually striking. In the end, we've settled for snow accumulation as the main focus and chosen an interaction method that would be at least a bit usable in video games.

The method we are focusing on was proposed by D. T. Reynolds, S. D. Laycock & A. M. Day [[3]] in The Visual computer during Mr. Reynolds's doctorate studies. The goal was to create a method that allows real-time accumulation of snow in an ever-changing dynamic environment.

We have stumbled upon a few problems during the implementation of the method, so I have reached out to Dr. Reynolds to see if he could help. Not only did he send me his materials and source code, but he also scheduled a meeting with me to answer any questions I might have. We could not have finished this work if it was not for him.

4.1 Overview

The method described here utilizes many known techniques, such as shadow mapping, and modifies them to a certain extent to fit the needs. Each object in the scene is attached a single 2D accumulation texture which stores information about the snow height. First, the scene is rendered from the snowfall direction to determine which surface is not occluded and should receive snow. This information is then run through a single render pass utilizing geometry shaders to map the information from the occlusion render directly to all the appropriate accumulation textures. Snow stability and techniques such as random noise, dilation, and Gaussian blur further enhance detail and visual believability. Normal maps are calculated for each surface and used alongside dynamic tesselation to add a very high amount of detail. As this method focuses on real-time accumulation, no physics calculations are involved.

The graph below illustrates the entire process from start to finish. Each stage is explained in more detail in later sections.

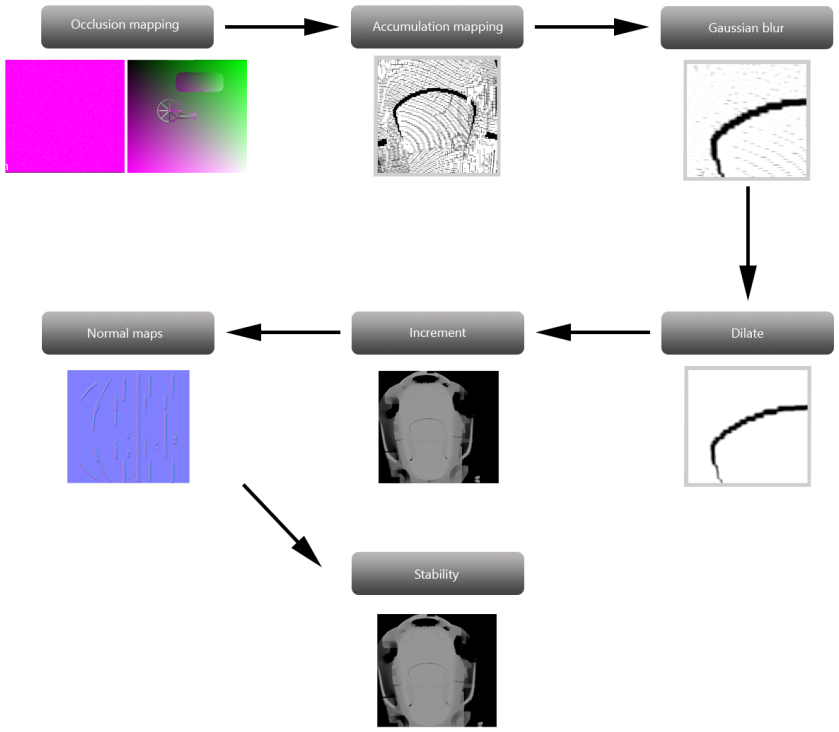


Figure 4.1: Graph showing all of the different stages involved

4.2 Accumulation phase

The first step and arguably the most important one is snow accumulation. The accumulation technique first proposed by the authors of this method allows for a real-time, scalable, and dynamic scene. Although, as we will explain in the later chapters, this doesn't allow for a very complex scene with many objects. One can take steps to achieve a higher scene complexity, but that would mean sacrificing on quality.

4.2.1 Occlusion mapping

The first step is rendering the scene from the position of snow source. Techniques described here are similar to deferred rendering or shadow mapping. Both of these widely used techniques involve taking a camera and rendering the whole scene from the light source position. In case of shadow mapping, since we are commonly rendering a scene with an included directional light, orthographic camera is of great use. The ouput of shadow mapping is a depth map (occlusion map) which tells us if a specific fragment is visible from the light's position. Deffered rendering in this case saves useful data about each visible object, such as their position, normals, color and so on.

Very similar approaches to those mentioned above are used in the proposed technique. We used an orthographic camera to simulate a directional light source in shadow mapping. This method is not too different. In a sense, all flakes fall in the same direction parallel to each other - similar to how light rays travel. This technique allows us to easily introduce strong wind forces and flake-flutter into the scene. Flake-flutter is achieved by introducing a minimal offset to the camera position and direction. Two RGBA color textures are output - contrary to outputting depth information in standard shadow mapping..

The first output texture stores the individual object's integer ID and a noise value. The ID is assigned to each object during it's loading phase as a pre-processing step. It's used to correctly pinpoint the accumulation texture of the object in question. The noise value is sampled from a Perlin noise texture with a variable offset to introduce a certain amount of randomness to the accumulation.

The second output contains an object's minimum and maximum texture

coordinates at the visible point. The built-in command used to sample the texture for the minimum and maximum coordinates is *interpolateAtOffset*. These unique texture coordinates are generated as part of a pre-processing step during the actual creation of an object - in Blender, for example. What is essential is that each face of an object has to be assigned precisely one unique non-overlapping face in the UV map. Allowing overlapping faces would mean two - or more - unrelated faces could receive the same amount of snow which is not a desirable side effect. The saved coordinates are then used to output the accumulation amount to a correct part of the accumulation texture. These accumulation textures hold all the required information about the snow height at each face of an object. Storing all the information in textures allows for a dynamic moving scene, as the information is saved in the map instead of being calculated at each frame from the beginning, for example, during an object's movement.

4.2.2 Accumulation mapping

With occlusion of the way, a separate render pass is run to process the occlusion textures to map accumulation to its corresponding texture. The pass is run for each pixel of an occlusion output, and while this might seem like a large and expensive operation, no complex matrix transformations are required. It consists of a simple pass-through vertex shader, a programmable geometry shader to do the necessary transformations, and a fragment shader with multiple render targets set to output each object's accumulation information. Very important to note is that blending has to be enabled for this stage to work correctly.

The input for the render pass consists of individual quads, whose vertex positions are set to arbitrary values and texture coordinates set to map each pixel of the occlusion render. The generation of this "grid" is not a complex problem and can easily be part of a pre-processing or an initialization step. These quads are sent through the rest of the render pipeline to result in quads covering one of the accumulation maps. The central part of this render pass is a geometry shader that handles the necessary transformations. More details can be found in the pseudocode below ((4.2.2)).

Vertex shader stage

Vertex shader here doesn't do much. Its only job is to handle outputting the provided information about the quad position so it can later be used in a geometry shader.

Geometry shader stage

The geometry shader is set to accept a single triangle per all of the quads, as triangle is the basic primitive a graphics pipeline is used to work with. Input of a geometry shader is an array of values because the geometry shader works with whole primitives rather than individual vertices. Provided texture coordinates from the previous pipeline stage are used to sample both of the occlusion textures in order to get the required information that is stored in them. These values are then saved into 4-dimensional vectors to be used later.

After getting the required information out of the occlusion maps, we can start calculating where within the individual occlusion maps shall we output information about the snow height. Once found, the triangle coordinates are transformed to result in a triangle covering an area of an accumulation map. More details may be found in the pseudocode below (4.2.2).

Fragment shader stage

Once the primitives with the correct coordinates are successfully transformed, they are sent through pipeline into the fragment shader. This fragment shader has multiple render targets which are the individual accumulation maps attached. The shader determines the correct texture to render to by an ID that is passed from the geometry shader. If blending wasn't enabled and set up correctly it would result in artifacts in places where individual object's UV coordinates overlap - leaving the area of one UV map empty.

Input: position, texcoord
Output: position, texcoord
Vertex Shader :
  $out : position \leftarrow in : position$
  $out : texcoord \leftarrow in : texcoord$
end

Input: position[3], texcoord[3]
Output: accumulation, texture_id
Geometry Shader:
  $ids \leftarrow occlusion1[texcoord[0]]$
  $mapping \leftarrow occlusion2[texcoord[0]]$

  vec2 $minPoint \leftarrow (map[0], map[1])$
  vec2 $maxPoint \leftarrow (map[2], map[3])$

  for $i = 0; i < 3;$ **do**

    if $position[i].x < 0$ **then**
      $glPosition.x \leftarrow minPoint.x$
    end
    else
      $glPosition.x \leftarrow maxPoint.x$
    end

    if $position[i].y < 0$ **then**
      $glPosition.x \leftarrow minPoint.x$
    end
    else
      $glPosition.y \leftarrow maxPoint.y$
    end

    $texture\_id \leftarrow ids[0]$
    $accumulation \leftarrow ids[1]$
  end
end

Input: accumulation, texture_id
Output: accumulation_buffers[8]
Fragment Shader:
  $accumulation\_buffers[texture\_id] \leftarrow accumulation$
end

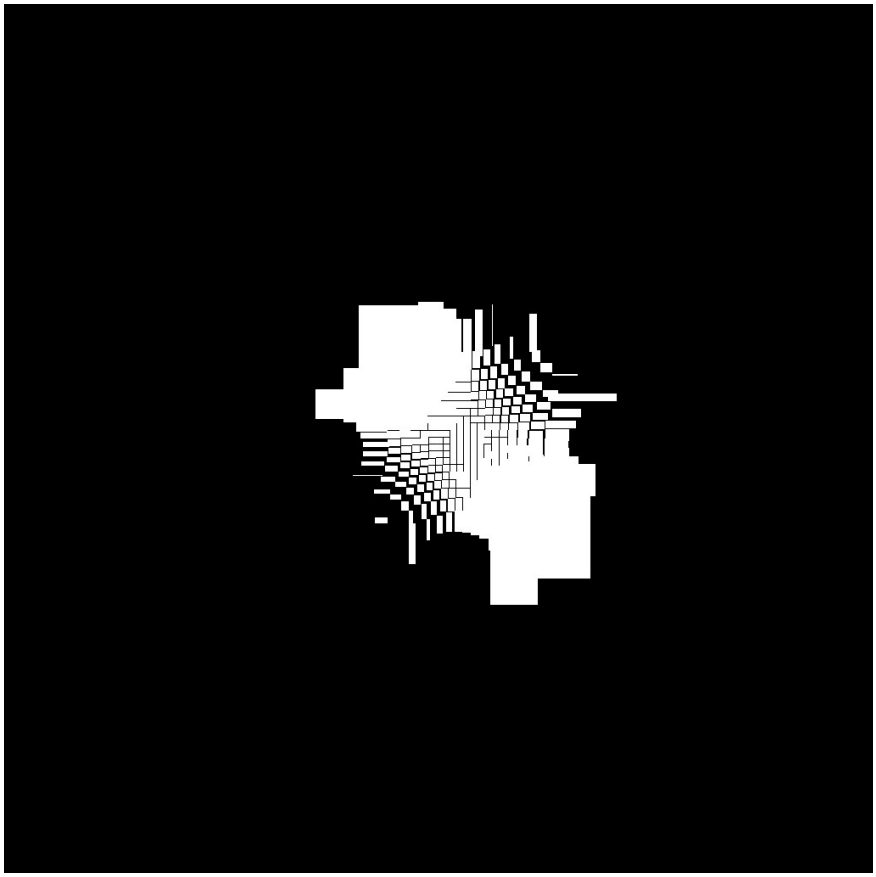


Figure 4.2: The result of the accumulation stage

4.2.3 Postprocessing

Now that all the accumulation information for our current frame has been rendered to it's corresponding buffer, we have to apply some more postprocessing in order to make it more realistic and believable. All of the different effects are of course their own separate render passes. This doesn't hinder the real-time performance even with high resolution occlusion and accumulation textures, albeit on a good testing equipment.

Almost all of the below mentioned postprocessing effects are in the original method. However, after further inspecting, I've decided to implement them in a more advanced way than originally proposed and also add an entirely new effect in order to enhance the overall results.

Gaussian blur

The first effect all of the accumulation maps go through is called Gaussian blur [[9]]. All the rendered quads have very hard edges and look too monotonous. Gaussian blur here is presented to introduce smoother edges and even transitions between the colors of the shapes and the background of the texture - which basically symbolizes zero accumulation.

As with many image processing methods, Gaussian blur too uses convolution [[10]] to process the image. A 5x5 kernel is used to process a every pixel of each texture to result in a smooth and blurred image. Offsets and weights are included below. [4.3]
offset = { 0.0, 1.0, 2.0, 3.0, 4.0 }
weights = { 0.2270270270, 0.1945945946, 0.1216216216, 0.0540540541, 0.0162162162 }

Figure 4.3: Accumulation map after applying Gaussian blur.[9]

Dilation

Dilation is an effect the original method [[11]] does not include, but we've decided to add (4.4). The original method, as previously seen on screenshots of accumulation maps, creates margin of various sizes between individual color quads, causing possible artifacts. By introducing dilation, these margins can be significantly reduced, potentially reducing artifacts. Although, after a bit of testing, we've noticed that dilating the textures is not always the best idea. While it works nicely for uniform solid objects, problems arise for non-uniform objects with spaces and holes in them, like a bench or a fence. These spaces should result in a grid-like pattern on the underlying surfaces. When we take a look at the accumulation textures, these holes are usually about a pixel wide. Given this size, dilation will result in a non-zero value in those spots. Being able to turn off dilation for certain objects might be a good idea.

Dilation uses a method similar to convolution, in that it uses a convolution matrix to do it's calculations. For every pixel of each texture, we look at the neighboring pixels in a defined area, determine which of the pixels has the highest per-element value and use that to replace the value of the current pixel. The area is defined in code, depending on the needs.

Increment

A render pass that doesn't need much explaining is the increment pass. It takes the values stored in the accumulation texture from the previous frame, the values calculated in the current pass and simply adds them together, resulting in over-time accumulated values. By making this a separate render pass, we can utilize the countless number of cores present on the GPU and greatly paralelize this algorithm, potentially finishing it in no time.

Normal maps

Lighting is one of the essential parts of any scene, as it also adds realism and details. In order to calculate lighting, we have to know something called a surface normal. A surface normal tells us which way the surface is facing. This is crucial in lighting calculation because it allows us to determine if a face is oriented towards the light or entirely in a different direction.

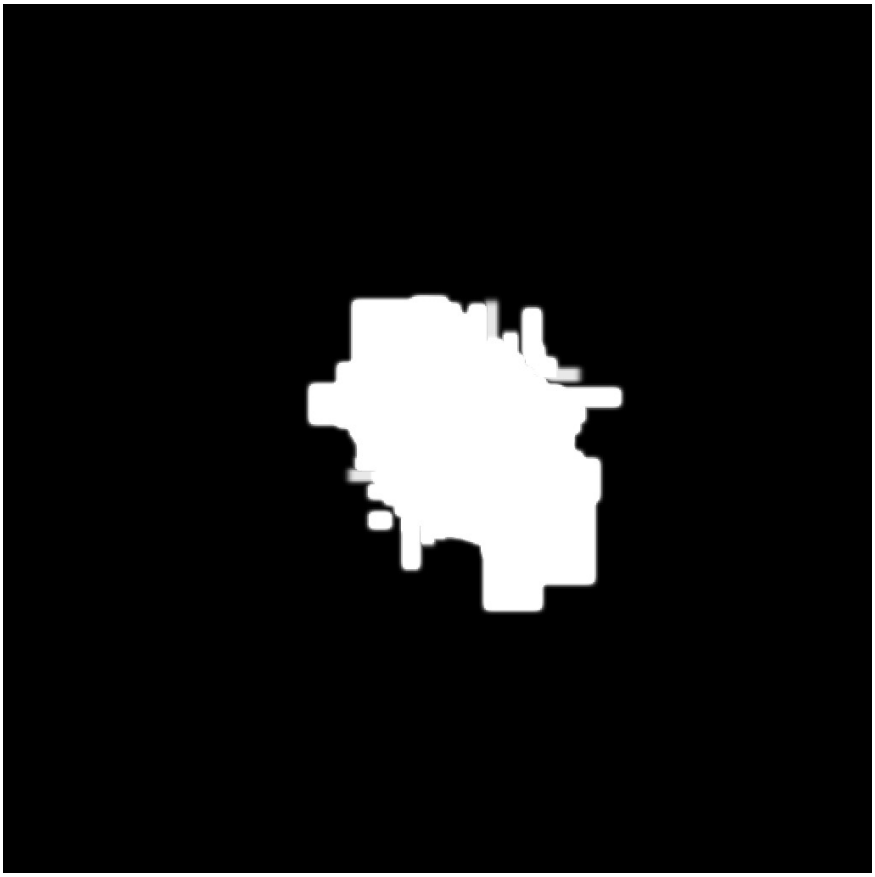


Figure 4.4: A dilated accumulation textures.

A commonly used technique utilizing normals is normal mapping, which involves saving the information about surface normals in a 2D texture used in lighting calculations. By changing the direction of per-surface normals and hence the actual direction of a surface, we can simulate a high level of detail without adding any geometry. Of course, the result will not be perfect, and from a certain perspective, it's going to be obvious what is going on, but despite that, normal maps are perfect for adding very fine detail.

Snow accumulation de facto changes the slope of the surface and, as such, causes the light to bounce off in a different direction. We can simulate this behavior by using normal maps. Since we know the height of each surface, we can use it to calculate the normal directions. Calculation of these normals is not too complex, and as with many image processing algorithms, it uses a convolution filter. This particular filter is called the Scharr filter ([[17]] and [14]). The X and Y values are calculated by adding the neighboring pixel values, each multiplied by coefficients defined by the filter.

$$\begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix} \qquad \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

Scharr-x **Scharr-y**

Figure 4.5: The Scharr filter for X and Y. [17]

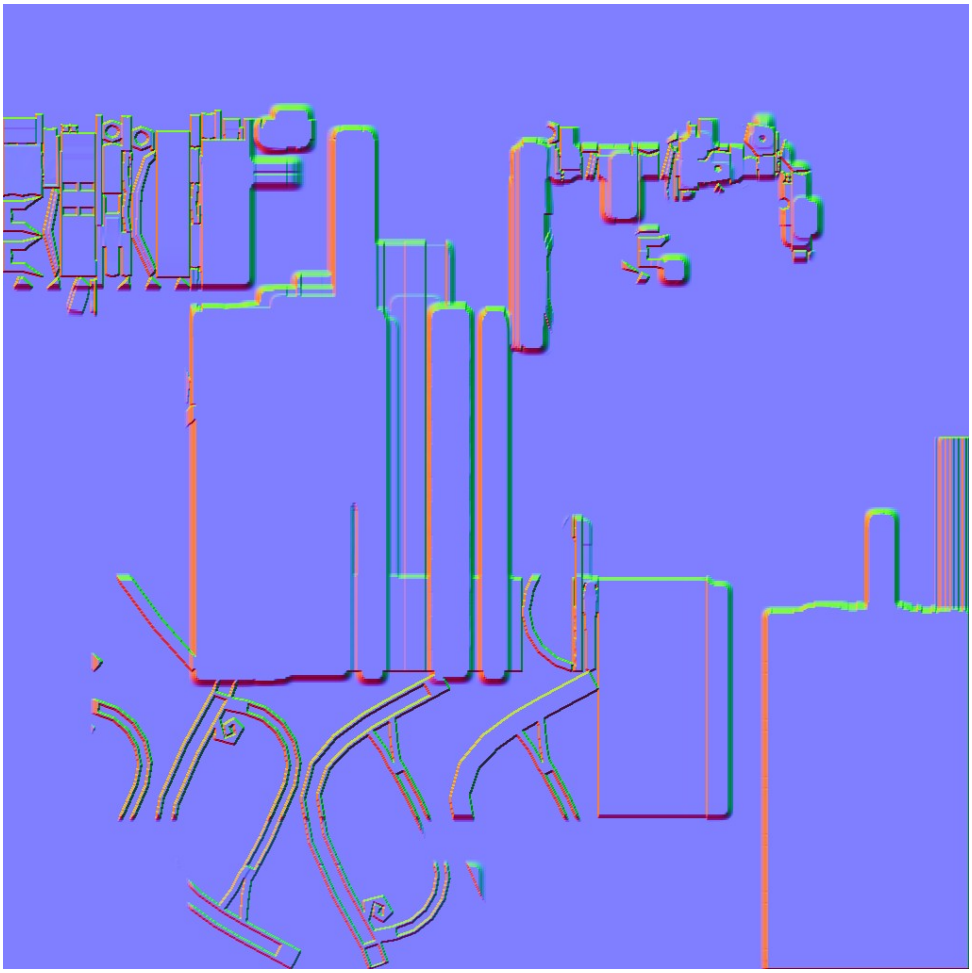


Figure 4.6: A sample normal map for a bench [14]

4.3 Stability

Just freely accumulating snow on visible surfaces isn't enough to produce visually believable, let alone realistic results (4.7). That is because snow can't just stick to any surface visible. Of course, under the correct conditions, certain amount of snow can stick even to very steep surfaces or even create bridge-like structures between two very close objects. However, taking into account all of the physical properties and condition for that to happen is impossible in case of a real-time simulation. The goal is to find a method that is simple enough to allow for real-time use cases that also produces realistic enough results. We have to ensure that the amount of snow retained by a surface changes depending on it's rotation and orientation (4.8).

This method simply introduces a falloff rate, at which the snow falls of a certain surface. Objects are passed along with geometry information, such as surface normals, in order to calculate the steepness - angle between the normal and an up vector (vector pointing towards the sky). Snow height at any point of the surface is determined by the equation below. The small letter $\boldsymbol{a}$ stands for current accumulation, $\boldsymbol{f}$ is short for falloff rate and is constrained: $\boldsymbol{1} \geq \boldsymbol{f} \geq \boldsymbol{0}$. Normal vector is expressed as small $\boldsymbol{n}$ and is used to calculate the angle between the surface and an up-vector $\boldsymbol{y}$ and $\boldsymbol{m}$ stands for "maximum snow height".

$$result = \begin{cases} a - (a - (\mathbf{n} * \mathbf{y}) * m) * f_r, & if(a - (\mathbf{n} * \mathbf{y} * m)) \geq 0 \\ 0 & otherwise \end{cases}$$

4.4 Dynamic tessellation

Since OpenGL version 4.X.X, developers can take advantage of a new Vertex Processing stage- tesselation [[7]]. Tessellation involves taking a patch of vertices and subdividing them into smaller primitives. Two new shader stages are introduced, the optional TCS - Tessellation control shader - which dictates how much tessellation should be performed, and the TES -Tessellation evaluation shader - which takes in a tessellated patch and calculates new per-vertex values. This allows us to dynamically create more geometric detail where needed and further enhance realism.

Although tessellation itself doesn't seem to affect performance all that

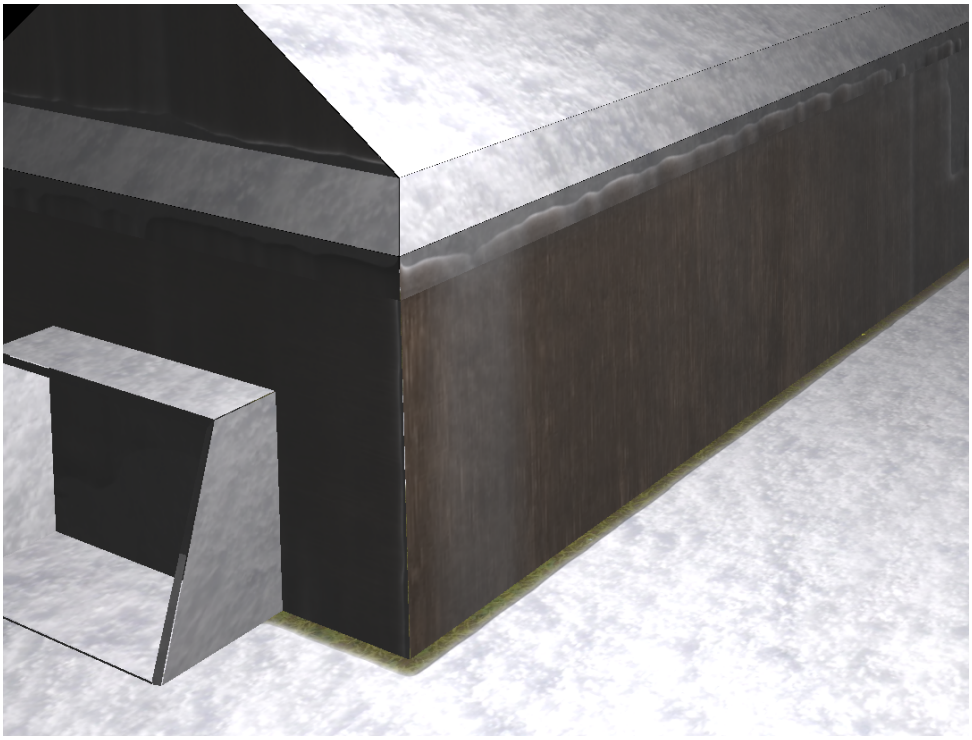


Figure 4.7: Cottage with turned off snow stability

much [[8]], we still shouldn't underestimate its effects. As stated in the [[8]] article itself, the FPS might drop by almost 25% in case of a higher resolution. That means we have to limit the amount of tessellation each object receives.

Not all objects in the scene might benefit from further subdividing their mesh, either because it's not able to hold that much snow or because it already has enough geometric detail making tessellation obsolete. Distance from the camera is also a very good point to consider when deciding how much to tesselate an object or if to tessellate it at all.

Our implementation calculates the amount of tessellation depending on the distance of an object from the main camera. The further away an object is, the lower the tessellation setting. Tessellation can also be turned off by passing a boolean value through a uniform object.

The distance of an object can be determined easily by looking at the **z** coordinate in the View space. This value is then multiplied by a maximum tessellation factor and clamped between the values 0 and 1. The resulting float number represents the interpolation ratio between the maximum tessellation allowed, which is 64, and the lowest - which depends on the patch primitive type.

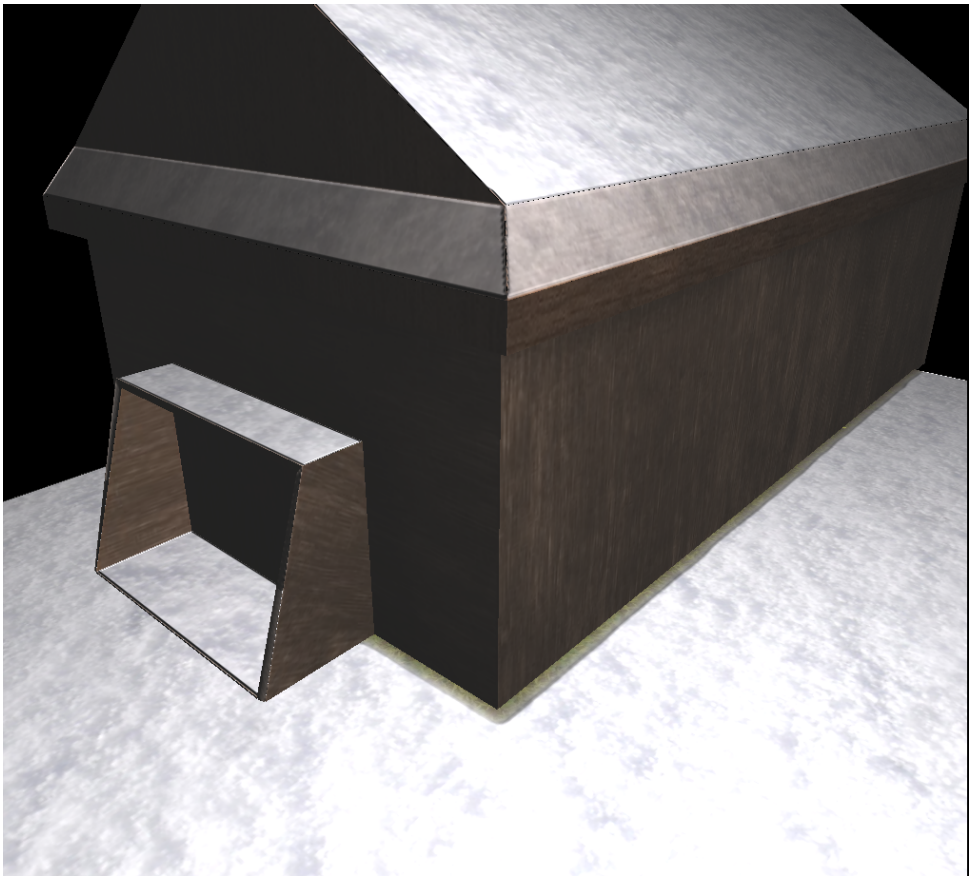


Figure 4.8: Cottage with turned on snow stability

The tessellation evaluation shader then handles the interpolation of all the different input values, like vertex position, normals, tangents, and more. It does that by transforming the corresponding coordinates using barycentric coordinates provided by the built-in **gl_TessCoord** variable. It is also where vertex displacement based on the snow height happens. An example interpolation of a vertex position is shown in an equation below.

$$u = gl\_TessCoord.x;$$
$$v = gl\_TessCoord.y;$$
$$w = gl\_TessCoord.z;$$

$$tesPosition = u * inPosition[0] + v * inPosition[1] + w * inPosition[2];$$

Chapter 5

Snow deformation

Since the goal of this thesis was to create a dynamic snow surface, along with snow accumulation, at least a simple snow deformation method of the ground had to be implemented. Even so, it's of utmost importance to emphasize that the main focus of this method was to implement accumulation. Due to this fact, deformation doesn't get much focus in this text.

Daniel Hanák introduced a good deformation method in his master's thesis [[6]]. It uses an orthographic camera to capture all the objects capable of affecting the snow surface. The result of this capture is a heightmap that is then used in other calculations to determine how much an object should deform the surface.

The method implemented here is a major simplification of the method proposed in the thesis. It simply captures all the deforming objects in question from below using an orthographic camera, transforming them to View space and finally outputting their distance value, which is transformed and clamped to fit in the range from 0 to 1. The distance value is the fragment's position's **z** coordinate. Each output value represents the amount of snow to be removed from the ground's surface at that location.

The process of applying the deformation is almost identical to the method already described here [4.2.3]. In this case, the values are instead subtracted from each other.

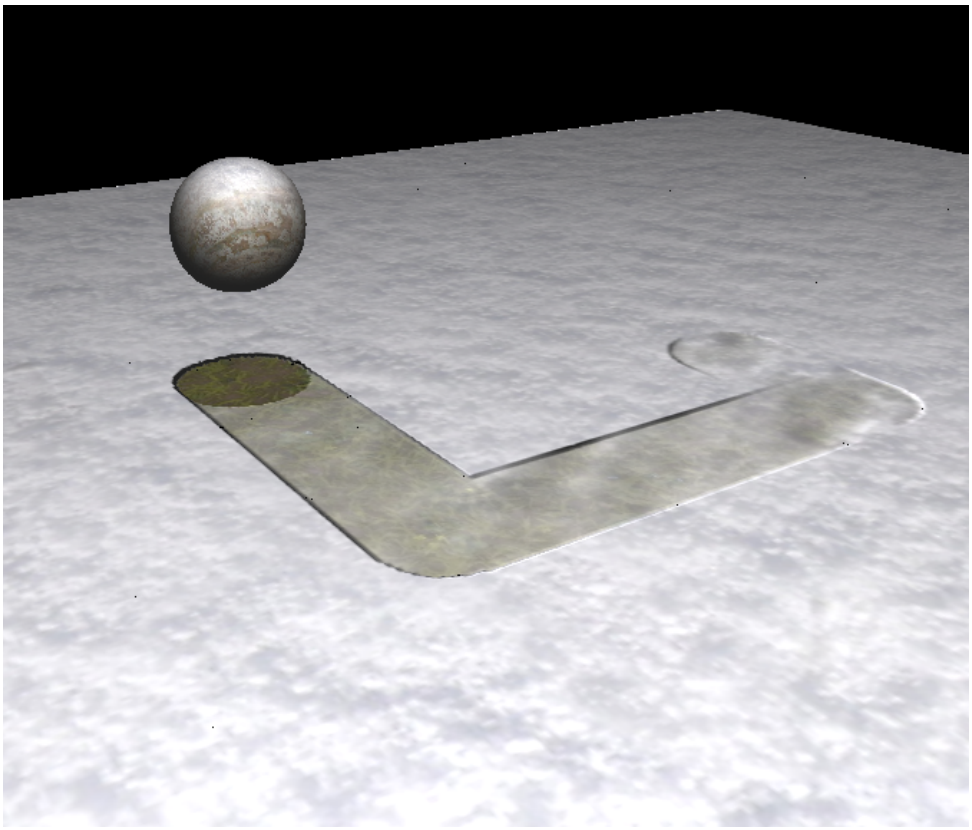


Figure 5.1: Snow accumulating in spots previously deformed

Chapter 6

Results

The last step is to test out the application. We have to take a closer look at the results to determine how successful we were in implementing the method and how useful it is in real-time environments.

6.1 Rendered scenes

One of the scenes shown in the original paper is a scene containing a bench. I've tried my best to find a bench that is similar to the bench in the paper to keep the comparison relatively fair. Then, I've imported the model into Blender to fine tune the mesh. The goal was to fix topology, set smooth shading, join meshes together etc. Doing these will result in a better looking accumulation and tessellation.

Here are two screenshots, each depicting a different stage of accumulation. I believe both of these pictures show the effects of normal mapping and also dynamic tessellation. Tessellation is visible the most in the first picture, while the second picture better shows the power of normal mapping, thanks to which we can see a division between surfaces with and without snow (6.1). You can see the backrest not accumulating much snow. That is due to it's rather steep angle which means the stability algorithm will mark the spots for deletion.

Figure 6.1: Later stage of accumulation, showing the power of normal mapping

Next are two images to compare our results against. The goal of this method is, of course, to simulate the real world as closely as possible. To determine how successful we were, we have to compare the results with a photo from the real world (6.2). The second image (6.3) is a screenshot taken from the original author's paper to see how our implementation differs.

Figure 6.2: Real world bench [15]

Figure 6.3: Bench from the original method [3]

Compared to the original method, our bench has fewer noticeable artifacts along the top of the bench. Overall, the accumulation also seems a bit

smoother - it looks as if the original scene used lower resolution textures.

The real-world example is almost impossible to achieve with this method due to the absence of physics calculations. We would have to consider uncountable factors to achieve a result similar to the example—the properties of snow, the bench's material composition, etcetera.

Figure 6.4: Another comparison with a real-life example. Mustang taken from [[13]]

Another scene we have been testing this method on is a scene containing just the ground and a giant lion head (6.5 and 6.6) with tessellation turned on. The head itself contains more than a hundred thousand triangles. We can see that this method works quite well, even on objects with higher levels of complexity. Of course, the results are not perfect. Especially the diffuse textures appear to be blurred - most probably due to the UV unwrapping. Besides that, the results are more than satisfying. There do not appear to be too many artifacts; tessellation and displacement produce a good-looking mesh. Despite the high complexity, occlusion seems to be working well, and even the tiniest potentially occluded spots do not seem to hold much snow.

The lion head is also perfect for showing why a good quality UV unwrapping is an absolute necessity. This UV was produced by Blender's built-in "Smart UV Project" function. It split the mesh into countless little islands, where each neighboring island may be from a completely different part of an object. The consequence of this fact is illustrated by [6.7].

We cannot forget about testing scenes where snow does not fall precisely from above. The first scene contains a bench and a car chassis (6.8). Snow gets accumulated accurately; the right side of the car is entirely blank, with just a few snowflakes floating astray - as expected. Islands without any

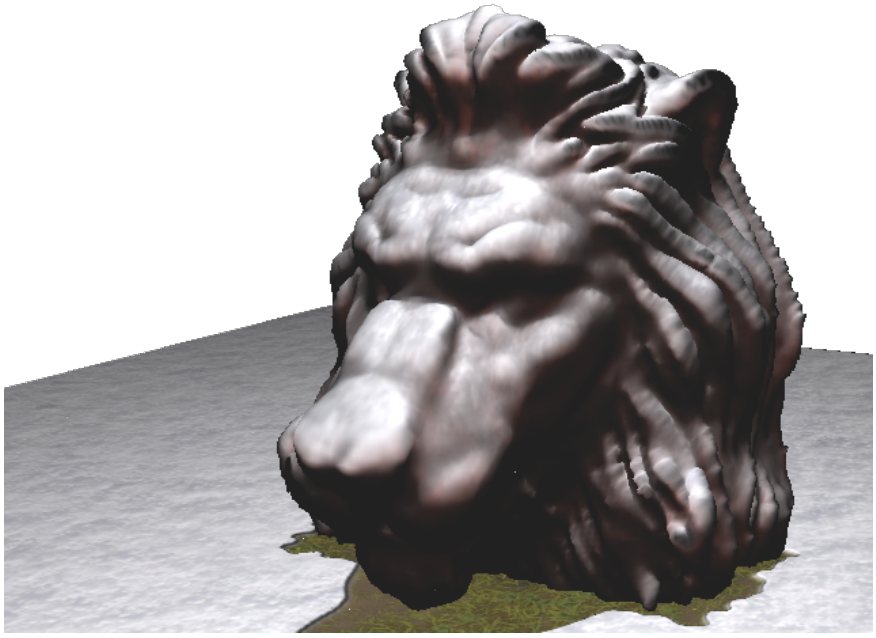


Figure 6.5: Front view of the lion head [2]

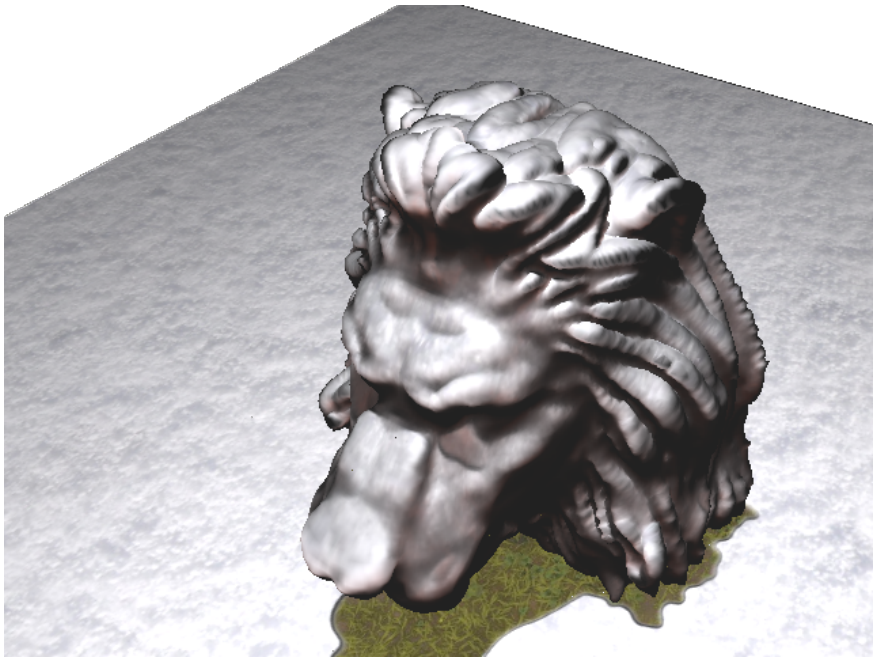


Figure 6.6: Top view of the lion head [2]

accumulated snow were created due to wind direction, similar to how shadows are created.

The image (6.9) shows a scene similar to the first one where the bench is

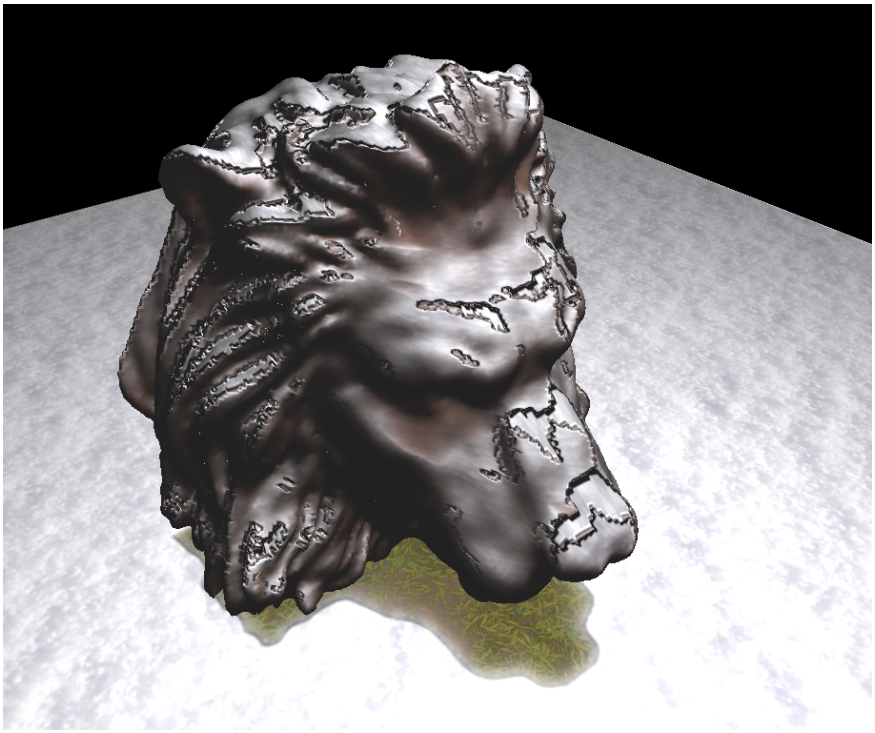


Figure 6.7: Lion head with a bad UV unwrap

moved closer to the car. This was done to illustrate further that occlusion works as expected. Although the top part of the bench seems to be accumulating a small amount of snow, that is simply due to the fact that it is visible from the snow source point of view.

One of the ways to control visual believability in computer graphics, for example, in games, is to increase the texture resolution. It makes sense - by increasing the resolution, we can dramatically increase the amount of information we can store. Just for comparison, a 512px wide texture contains 262 144 pixels, 1024px texture has 1 048 576 pixels while a 2048px texture contains a whopping 4 194 304 pixels!

While this might dramatically impact the overall visual quality and detail in other areas, this method doesn't gain much by drastically increasing the texture resolution. We can see that clearly in the images below. The 512px wide texture produces visible grid-like artifacts (6.10) that might hinder the overall experience, but the differences between the 1024px (6.11) and 2048px (6.12) are almost non-existent. It might seem that using the 1024px is the ideal choice not only due to the visual quality. In the later section, we will take a look at performance.

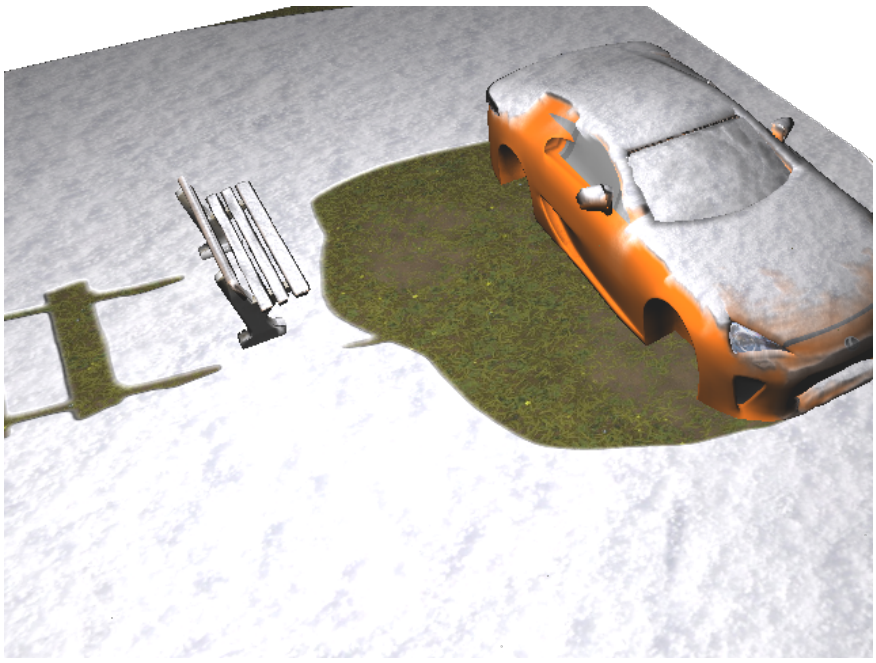


Figure 6.8: Scene where snow falls from the side

Figure 6.9: As we can see, the bench accumulates snow only in the upper part

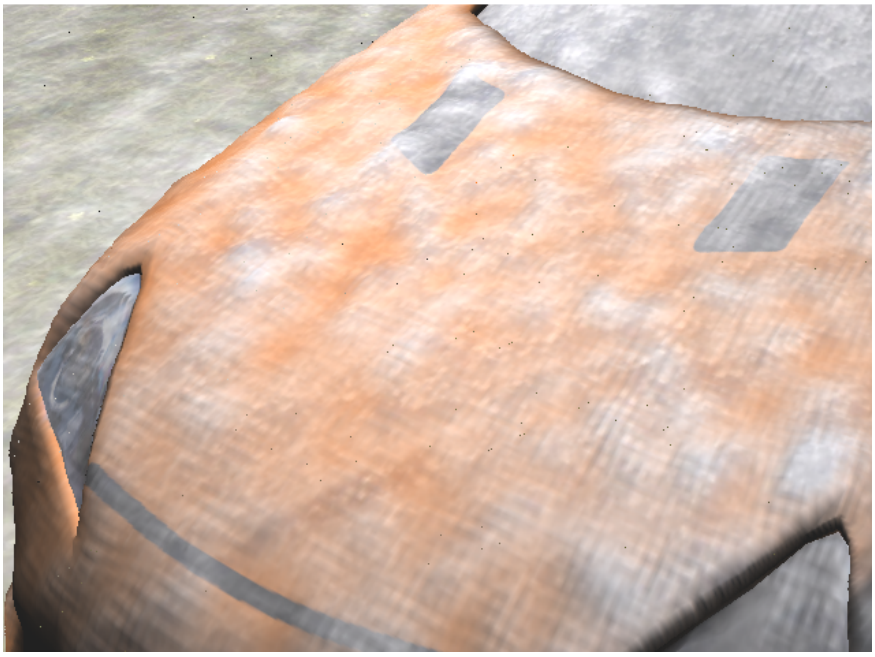


Figure 6.10: The artifacts present when using a 512px texture

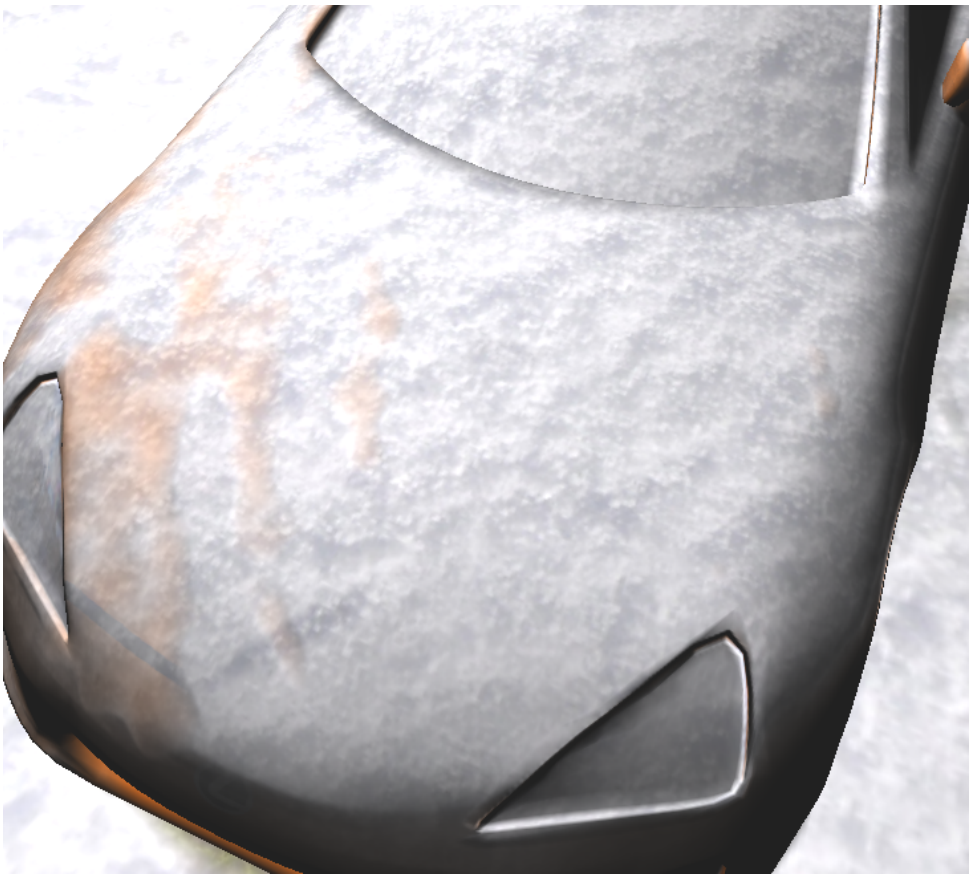


Figure 6.11: Accumulation when using the 1024px texture

6.2 Profiling

While visuals are a crucial part of any snow simulation method - or any method in computer graphics in general - we cannot forget about performance, even more so when developing a method for real-time use cases. For that reason, we've decided to test this method against multiple scenarios and compare the results. All of the test were run on a computer with AMD Ryzen 7 5800H, Nvidia GeForce RTX 3060 with 6 GB of VRAM and 32 GB of RAM.

We've created three scenes of varying complexity to test the method. The tests consist of running each scene with different parameters, such as texture resolution, in a special profiling environment (Nvidia NSIGHT [[12]]) - to get information about the duration of each draw call. Each test case was being run with varying texture resolutions - namely 512px, 1024px and 2048px. The results we are interested in are: accumulation pass duration, duration of individual support passes (blur, dilation, normal maps...), frames per second and memory consumption.

We've begun our testing with a scene consisting of eight objects with a very simple mesh - that means a mesh with a small number of triangles.

The second scene is a forest consisting of just two objects - the ground and the forest mesh introduced in the previous chapter. Despite containing just two distinct objects, the forest mesh alone has over 360 thousand triangles, making it an ideal candidate to compare the previous case against. More on that later.

And finally the third scene, where we took the second scene and added six more objects. Each of these objects is a tree mesh extracted from the forest mesh. This last test case is a defacto combination of the previous two, making it ideal for pushing the method to its limits. [6.13]

The table below shows the results.

	Texture resolution	Accumulation pass	Support pass	FPS	Memory consumption
Scenario 1	512px	0.22ms	0.12ms	72fps	300MB
	1024px	3.58ms	0.37ms	35fps	1100MB
	2048px	42.68ms	1.39ms	4.9fps	3100MB
Scenario 2	512px	0.26ms	0.14ms	144fps	200MB
	1024px	0.9ms	0.37ms	127fps - 140fps	500MB
	2048px	9.65ms	1.39ms	20fps	1150MB
Scenario 3	512px	0.23ms	0.12ms	75fps	400MB
	1024px	2.33ms	0.37ms	36fps	1000MB
	2048px	54.24ms	1.39ms	5fps	2900MB

Now an obvious question arises as to why tessellation does not play any role in the testing? Although it adds quite a bit of fine detail and enhances the overall visual experience, it's not at the core of the method, and also, the results are obvious. The more an object is tesselated, the longer it will take to render. The main goal of this testing is to show that even without tessellation, this method can slow down even a high-end computer.

One may ask, how is it possible that the stress-testing third scenario is comparable to the first. Although not immediately apparent, the answer to that question is simple. We have to realize that each support calculation is a separate render pass. Although each pass might take just over 1ms, if you multiply that number by about 40 - five render passes each doing calculations on eight 2048px textures - you get a non-trivial render time. To illustrate that better, if we multiply 1.37ms by 40, we get 52ms. A frame time of 52ms equals roughly about 20 FPS! And if you add the time it takes to calculate the accumulation maps, pass the data to the GPU or render the whole scene, you get a high frame time.

We can see here that to achieve a stable framerate while maintaining good visuals; the 1024px textures seem like the ideal choice. As we've seen in our previous testing, the difference in visuals between 1024px and 2048px textures is not very apparent. However, it may play a role if we unwrap many objects onto a single texture. Even though 512px textures provide the most significant boost in framerate, using them introduces noticeable visual artifacts as previously seen (6.10). It can also be seen that the jump in memory consumption is not as high between the 512px and 1024px texture resolutions.

We have decided not to include the deformation phase since it has little to no effect on the final framerate.

Figure 6.12: Accumulation when using the 2048px texture

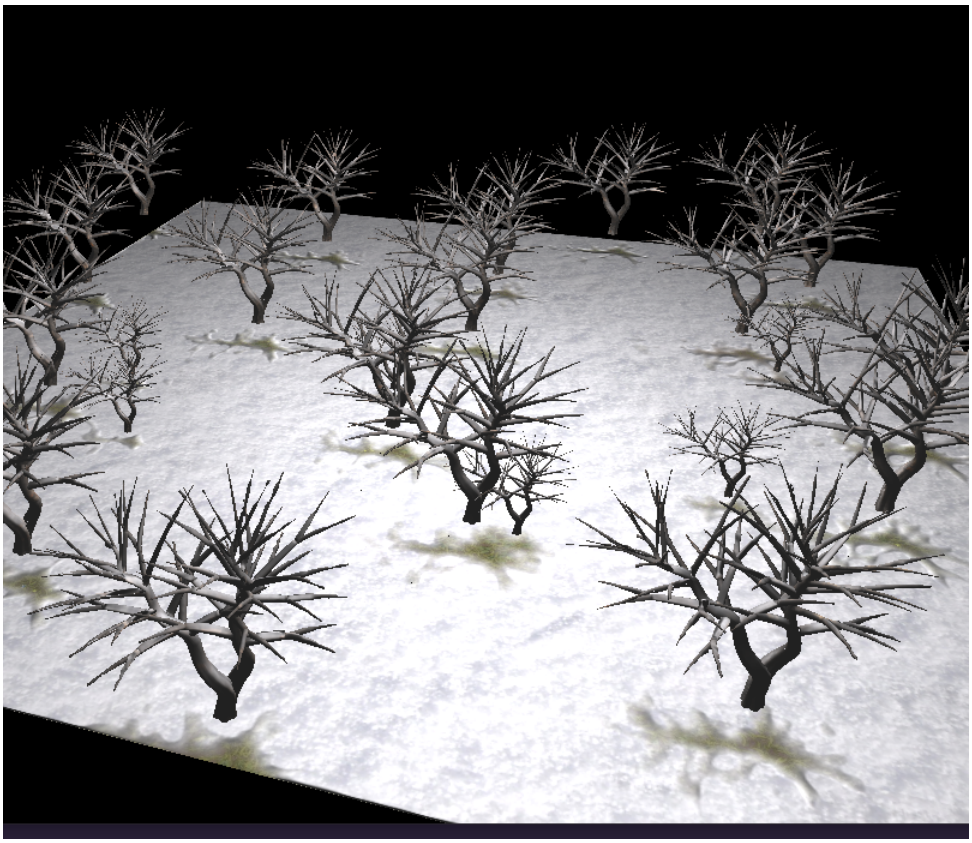


Figure 6.13: Scene containing the maximum amount of objects per render target along with a multi-mesh object containing many trees, unwrapped onto a single texture

Chapter 7

Conclusion

The main goal of this work was to do research on various methods of snow accumulation and implement and build upon the method proposed by D.T Reynolds [[3]]. We have stumbled upon many difficulties along the way, but thanks to Dr. Reynold, we solved those difficulties and successfully finished the application. It is fair to say that many algorithms that implement parts of the original method, such as normal map generation, were upgraded, resulting in, arguably, a better visual representation, as seen in the 6.3).

Next in line was snow deformation. Although, it was not the primary goal of this work. Consequently, only a small section of this thesis focuses on it. Despite not being the main focus, the result is good enough, and it fits the whole application well.

Many examples of scenes were provided and tested for performance (6.1) , which clearly shows that the method is very usable in a real-time environment, possibly even used in a simple game.

Appendix A

Bibliography

[1] Assimp. The model loading library installation guide. *https://github.com/assimp/assimp/blob/master/Build.md*.

[2] CGTRADER. Lion pendant download origin site. *https://www.cgtrader.com/*.

[3] D.T.Reynolds. Real-time accumulation of occlusion based snow. *https://link.springer.com/article/10.1007/s00371-014-0995-5*.

[4] Paul Fearing. Computer modelling of fallen snow. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp.37-46*, 2000.

[5] Beneš B Foldes, D. Occlusion-based snow accumulation simulation. *The Fourth Workshop on Virtual Reality Interactions and Physical Simulation, pp. 35–41*, 2007.

[6] Daniel Hanák. Real-time snow deformation. *https://is.muni.cz/th/m2v6i/real-time$_s$now$_d$eformation.pdf/*.

[7] Khronos. Tessellation. *https://www.khronos.org/opengl/wiki/Tessellation/*.

[8] Lambros Koukis. Effect of tessellation on fps. *https://c4re.gr/does-tessellation-affect-fps-performance/*.

[9] learnopengl. Advanced lighting - bloom. *https://learnopengl.com/Advanced-Lighting/Bloom*.

[10] Multiple. Convolution. *https://en.wikipedia.org/wiki/Convolution*.

[11] Multiple. Dilation. *https://en.wikipedia.org/wiki/Dilation$_($morphology$)$*.

[12] NVidia. Nvidia nsight graphics. *https://developer.nvidia.com/nsight-graphics*.

[13] Unsplash Oleksii S. Ford mustang in snow. *https://unsplash.com/photos/CLL5kUQHrW8/*.

[14] C. Petry. Generation of normal maps online. *https://github.com/cpetry/NormalMap-Online*.

[15] freestock.org Pexels. Wooden bench. *https://www.pexels.com/photo/brown-wooden-patio-bench-on-snow-288393/*.

[16] Christian Markowicz Ali Hassan Prashant Goswami. Real-time particle-based snow simulation on the gpu. *https://bth.diva-portal.org/smash/get/diva2:1320769/FULLTEXT01.pdf*.

[17] The AI Learner site. The scharr operator. *https://theailearner.com/tag/scharr-operator/*.

[18] Per Ohlsson Stefan Seipel. Real-time rendering of accumulated snow. *https://www.researchgate.net/publication/249762382*$_{R}eal - time_{R}endering_{o}f_{A}ccumulated_{S}now, January$2004.

[19] K. Tokoi. A shadow buffer technique for simulating snow-covered shapes. *Proceedings of the International Conference on Computer Graphics, Imaging and Visualisation, pp. 310–316)*, 2006.

Appendix B

Short tutorial on how to use the application

B.1 prerequisites

The application runs on 64-bits. In order to run the application, you have to have certain libraries installed. These include: GLFW, ASSIMP (installation instructions: [1]).The GLFW library should already be present in the solution directory.

B.2 Before running the application

The scene is, by default, pre-loaded with the forest scene with tessellation turned on by default. Each gameobject has a property called *should_tesselate* that turns it on and off for each individual object.

To load a different "scene", please refer to the *initGameObjects()* function. Although not very intuitive, you can define your own scene composition by commenting out objects. One thing to note, only 8 objects can be present in the **gameObjects** vector (the maximum number of render targets).

a) initTestTrees already adds 6 independent objects to the scene. b) by allowing deformation (*initDeformationMap* and **DeformationPhase**), you

add another object into the scene

B.2.1 Movement

Movement is a classic WASD, where **W** is front, **S** is back, **A** is left and **D** is right. You can look around the scene using your mouse.

B.2.2 Deforming

If you want to deform the surface of the ground in real time, just leave the *initDeformationMap* and **DeformationPhase** functions uncommented. Then, in the scene, just press the **P** key on your keayboard to switch between camera movement and object movement.