Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2019

Volume-Preserving Deformation of Terrain in Real-Time

Jesper Persson

Master of Science Thesis in Electrical Engineering

Volume-Preserving Deformation of Terrain in Real-Time

Jesper Persson

LiTH-ISY-EX-ET–19/5207–SE

Supervisor: **Harald Nautsch**
ISY, Linköpings universitet

Examiner: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Division of Information Coding
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

Deformation of terrain is a natural component in real life. A car driving over a muddy area creates deep trails as the mud gives room to the tires. A person running across a snow-covered field causes the snow to deform to the shape of the feet. However, when these types of interactions between terrain and objects are modelled and rendered in real-time computer graphics applications, cheap approximations such as texture splatting is commonly used. This lack of realism not only looks poor to the viewer, it can also cause information to get lost and be a barrier to immersion.

This thesis proposes an efficient system for permanent terrain deformations in real-time. In a volume-preserving manner, the ground material is displaced and animated as objects of arbitrary shapes intersect the terrain. Recent features of GPUs are taken advantage of to achieve high enough performance for the system to be used in real-time applications such as games and simulators.

Acknowledgments

I would like to thank my examiner, Ingemar Ragnemalm, at Linköping University for his valuable feedback and assistant throughout the project. I would also like to thank my opponent, Sebastian Andersson, for his comments on the thesis.

Linköping, June 2019
Jesper Persson

Contents

1

Introduction

Terrain is a common component in many video games and simulations. Additionally, there are often objects interacting with the terrain, such as a person walking or a vehicle driving. In real life, these types of interactions cause the terrain to deform. A vehicle driving on mud would create tracks from the tires, and possibly cause a ridge to build up along the track as mud is being forced upwards. A person walking on snow or sand would create visible foot trails. Depending on the properties of the material, feet would either cause rigid deformations, or have the material fall back to its initial state as the person lifts his foot back up.

Yet, when terrain is modelled and rendered, interactions often lack the dynamic deformations seen in nature. Instead, cheap approximations are often used such as splatting a texture after a vehicle to give the illusion of trails.

Lack of realism in the interaction between objects and terrain can cause some information to get lost. Footsteps and vehicle trails provide many hints to the viewer. As for footsteps, they convey how many persons has been at a certain place, what direction they went and the fact that they were even there. Deformations can also hint what object interacted with the terrain and the material of the terrain, since different materials deform differently. Furthermore, lack of realism can be a barrier to immersion in a video game or a simulation.

Deformation of terrain has been tackled in many projects. Sumner et al. [16] developed an offline algorithm for dynamic terrain deformation that showed great visual results. However, it runs on the CPU and is not tailored to real-time applications. In *Batman: Arkham Origins* [3], a snow covered scene allowed characters to walk, slide and fall on the snow, whereafter the terrain deformed to the shape of the intersecting object. This real-time application made use of the depth buffer to efficiently find intersections between characters and the snow. However, the in-

tersected snow was simply removed from the terrain, instead of being displaced and animated as was done in the algorithm by Sumner et al.

1.1 Problem formulation

This thesis aims to develop an efficient system for terrain deformations of arbitrary shapes in real-time that can be used in areas such as games, simulators and movies.

More specifically, the aim is to implement a parallel version of the algorithm by Sumner et al. [16] that makes use of modern GPUs to achieve real-time performance. Furthermore, the efficient depth buffer-based collision detection successfully used in [3] will be implemented to replace the ray-casting based collision detection used in [16]. The thesis also aims to address some limitations of [16], including its uniform displacement of material that doesn't take the direction and speed of intersecting objects into account when deforming the terrain. Lastly, the system will be evaluated to determine its performance and whether it is feasible to be used in a real-time computer graphics application.

1.2 Delimitations

The focus of the thesis is not to achieve deformations of terrain that adheres to the laws of physics. Instead, the goal is to create deformations in a visually convincing manner. Furthermore, the collision detection mechanism is based on rendering objects from below and analyzing the corresponding depth buffer. This limits the set of possible collisions, as the collision system merely knows what objects look like from below.

1.3 Structure

The thesis is divided into *introduction*, *theory*, *related work*, *method and implementation*, *result*, *discussion* and *conclusion*. The theory chapter explains necessary concepts in computer graphics. This is followed by a related work chapter that provides an overview of previous work in the area of terrain deformation. In the method and implementation chapter, the approach to terrain deformation in this project is presented. The result chapter provides an evaluation of the proposed system. After than follows a discussion chapter that elaborates on the proposed system in terms of how it performs and some ideas for future work. Finally, the thesis is summarized in the conclusion chapter.

2
Theory

This chapter presents concepts that are relevant to the thesis. This includes some background related to OpenGL and computer graphics, general-purpose computing on graphics processing units (GPGPU) and terrain rendering.

2.1 OpenGL

OpenGL is a popular API for accessing features in graphics hardware. It is commonly used to create images from models, which is referred to as *rendering*. *Models*, in this context, refer to objects consisting of geometry such as triangles, lines or points. The geometry is defined by its *vertices*. Each vertex is a point in space, that may contain additional data, such as color information [14, Ch. 1].

2.1.1 Homogeneous coordinate systems

It is natural to specify geometry in three-dimensional Cartesian coordinates. However, during rendering, OpenGL expects four-dimensional homogeneous coordinates. A three-component Cartesian coordinate (x, y, z) can be written as a four-dimensional homogeneous coordinate (xw, yw, zw, w). Thus, dividing all components of a homogeneous coordinate by the forth component yields the corresponding three-component Cartesian coordinate [14, pp. 216].

2.1.2 Rendering pipeline in OpenGL

The rendering pipeline in OpenGL is a set of processing stages that OpenGL uses to convert the geometric data the user application provides into a complete rendered image. The stages as of OpenGL version 4.3 are shown in Figure 2.1 [14, Ch. 1].

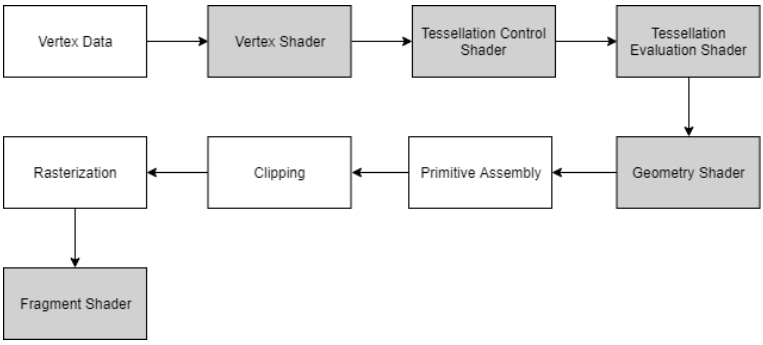


Figure 2.1: *Overview of the rendering pipeline in OpenGL 4.3.*

Before rendering, it is necessary to provide the vertex data to the GPU. *glBufferData()* can be used to move vertex data from main memory to video memory. Commonly, the vertex data is stored in a set of *vertex buffer objects* (VBOs) on the GPU.

Once vertex data have been moved to the video memory, the model can be rendered by calling one of OpenGL's rendering commands, e.g. *glDrawArrays()*. This will cause a series of *shaders* to be invoked. Shaders are small programs that are compiled by OpenGL and executed on the GPU. The first shader is the *vertex shader*. It is invoked for each vertex and is commonly used to transform vertices. The vertex shader is followed by an optional *tessellation shader* stage (which consists of two shader programs) and an optional *geometry shader* stage. Both of these can be used to generate additional geometry to produce higher triangle density. This can be used to reduces the bandwidth during rendering.

At this point, the vertex processing is completed. OpenGL now expects all vertices to be expression as four-dimensional homogeneous coordinates in *clip space*. Clip space refers to a coordinate space where coordinates whose X and Y component is within the range $[-1.0, 1.0]$ and whose Z component is within the range $[0, 1.0]$ will be rendered. Any geometry that falls outside this range will be discarded. However, before this happens, the primitive assembly stage takes the stream of processed vertices and combines them to base primitives, such as triangles.

Next, OpenGL performs *perspective divide*, which transforms the homogeneous coordinates to three-component Cartesian coordinates by dividing with the w component. These resulting coordinates are referred to as *normalized device coordinates* (NDC). The clipping stage uses the NDCs to discard triangles outside the clip space.

After clipping, OpenGL performs a viewport transform on the (X, Y) coordinate, and a depth range transform on the Z coordinate. By default, the depth range is between 0 and 1. The viewport is specified by calling *glViewport()* from the main application.

Next, the rasterizer generates fragments from the primitives. Each fragment is mapped to a position in the currently bound *framebuffer*. But before the fragment is written to the framebuffer it is processed by a *fragment shader*. While the previous user-programmable shader stages compute the final locations of vertices, the fragment shader computes the final color. This can for instance be achieved by reading from a texture or setting a static color. A fragment shader may stop the fragment from being drawn to the framebuffer by issuing a *fragment discard*.

Next follows some per-fragment operations that are used to decide whether the fragment should be written to the framebuffer or not. This includes *depth testing* and *stencil testing*. Depth testing is used to avoid fragments farther away from overwriting fragments whose depth are closer to the viewpoint. If the depth test succeeds, the depth buffer is updated with the new fragment's depth value to be used in future depth testing. The stencil test can be used for masking, by comparing the fragment's location with a location in the stencil buffer.

2.1.3 Off-screen rendering

Most commonly, OpenGL is used to render geometry to a screen. However, it is also possible to render to a different buffer. This has many use cases, such as shadow mapping, image filtering and GPGPU operations. This section explains the theory behind off-screen rendering.

A *Framebuffer Object* (FBO) is an object type in OpenGL that allows the user to create custom framebuffers. This enables off-screen-rendering, which means that OpenGL renders to a framebuffer that doesn't get drawn to the screen. A framebuffer is allocated with *glGenFramebuffers()* and bound with *glBindFramebuffer()*. To make the framebuffer useful it needs *framebuffer attachments*. One type of framebuffer attachment is a *renderbuffer*. They contain formatted image data. Similar to a framebuffer, a renderbuffer is generated by calling *glGenRenderbuffers()* and bound with *glBindRenderbuffer()*. The renderbuffer can be used as a color buffer, depth buffer or a stencil buffer. This is determined by the *internal format* parameter that is specified when calling *glRenderbufferStorage()*. The renderbuffer is attached to the framebuffer by calling *glFramebufferRenderbuffer()* [14, pp. 180-197].

Aside from rendering to renderbuffers, it is possible to render to texture maps. This can be useful when updating a texture that will later be used during rendering to the screen. *glFramebufferTexture()* is used to attach a texture to a framebuffer. It takes a parameter *attachment* that specifies if the texture should be used as a color, depth or stencil texture [14, p. 351].

Occasionally, it is desirable to read and write to the same texture, for instance during image filtering. However, the practice of reading from a texture that is simultaneously bound as the current framebuffer's writing attachment causes undefined behavior [14, p. 351]. Instead, subsequently using the same texture for reading and rendering is achieved using the concept of *ping-ponging*. When ping-ponging, two equally sized framebuffers are used. One is used for reading and the other is used for writing. After each frame their roles are swapped. This cir-

cumvents the problem of using the same texture for reading and writing [14, p. 891].

2.1.4 General-Purpose computing on GPUs

GPUs have become a powerful tool for general-purpose computation. They are no longer purely used to render geometry to the screen. The wide-spread popularity of GPUs can be attributed to the massive speedups achieved when applying GPUs to data-parallel algorithms [10, Ch. 29, 31].

While GPUs have gotten higher clock speeds as well as more and tinier transistors, their chip size has grown. The main focus of the additional transistors that have been added to newer generations of GPUs has been on computations, not on memory. This has enabled GPUs to perform many computations in parallel. However, as chip sizes have increased, the cost of communication has increased too, relative to the cost of computation. Algorithms that take advantage of this fact have a high *arithmetic intensity*, which is the ratio between computation and bandwidth [10, Ch. 29, 31].

When performing general computations on GPUs it is common to represent data as 2D textures. Many operations, such as matrix algebra and image processing map well to this structure. Even if the underlaying data is not grid-like, it can still be represented in a texture, where each pixel can be seen as an element in an array. The massive potential speedup is achieved by having fragment shaders operate in parallel on different pixels. This is done by rendering a quadrilateral over the entire framebuffer. This causes the rasterizer to generate a fragment, and thus a fragment shader invocation, for each pixel. The fragment shader reads from a texture (possible many textures at multiple locations), performs some computation and then writes to an output texture.

Since a fragment shader can only write to a predetermined place on the bound FBO, this approach to computation is only applicable to gather-like algorithms.

2.1.5 Shader Storage Buffer Object

Shader Storage Buffer Object (SSBO) is an OpenGL object that allows shaders to write structured data to a buffer. Furthermore, it is possible to write to arbitrary locations of the buffer from a shader invocation [14, p. 576-577]. SSBOs support the use of atomic operations, such as *atomicAdd()*. This enables multiple fragment shader invocations to write to the same memory location in the buffer at the same time. The order of memory accesses is not guaranteed. But the operations are guaranteed not to be interrupted [14, p. 890].

2.1.6 Performance optimizations

To utilize GPUs efficiently, it is important to take arithmetic complexity into account. To this end, texture resolution, the number of channels and the bit depth should not be higher than necessary. Compact textures can save memory and increase data transfer rates [14, p. 858]. OpenGL provides a wide range of inter-

nal texture formats that allows the user to limit the bit depth and the number of channels.

As texture lookups require data communication, they should be minimized. However, some shader operations require texture sampling at multiple locations during a shader pass. OpenGL provides a built in function called *textureOffset()* for doing this more efficiently. Instead of manually calculating a new texture coordinate, this function allows the user to specify a texture coordinate and an offset. The offset must be a constant expression [14, p. 341].

2.2 Low-pass filter

One common image processing operation is low-pass filtering. This can be used to blur an image. Low-pass filtering can be achieved by repeatedly applying a convolution kernel to each pixel of an image in a shader program. Expression 2.1 shows an example of a 3 by 3 Gaussian kernel.

$$\frac{1}{16} \cdot \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.1}$$

For a shader pass, each pixel is processed in parallel by a fragment shader. The program samples the eight pixels surrounding a central pixel and the central pixel itself. Next, each pixel value is multiplied by the corresponding value in the convolution kernel. The values are added together and written to the output texture.

2.3 Distance transform

A distance transform [11] is an operation defined on binary images. A subset of the pixels in the input image are called *seeds*. The operation computes a new image such that each pixel stores the distance to its closets seed. There exists multiple distance metrics, such as Euclidian distance, Manhattan distance (also known as city block distance) and Chebyshev distance (also known as chessboard distance). In a two-dimensional space, these metrics are defined as $\sqrt{x^2 + y^2}$, $x+y$ and $max(x, y)$ respectively, where (x, y) is a directional vector between two points.

2.3.1 Jump Flooding

Jump Flooding [11] is an algorithm that can be used to calculate the distance transform of an image efficiently on the GPU. The algorithm requires $log_2 n$ shader passes, where n is the image dimension. As a preprocessing step, the seeds are written to the image by storing the seed's coordinates in the red and green channel. Next, a shader program is invoked $log_2 n$ times, each time passing a uniform variable with values $n/2, n/4, \ldots, 1$ which represent the sampling distance. The

fragment shader is invoked for each pixel. Given a sampling distance k, each pixel reads from eight surrounding pixels at locations $(x + i, y + j)$ where x and y is the current pixel's location, and $i, j \in \{-k, 0, k\}$. By iterating over the surrounding pixels, the current pixel finds which seed is the closest. The coordinates of the closest seed become the output of the current pixel. Figure 2.2 shows how a seed in the lower left corner propagates to all pixels. After $\log_2 n$ executions, each pixel stores the coordinate of the lower left pixel.

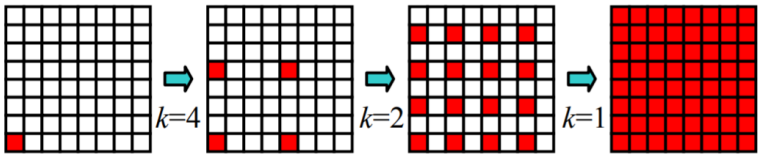


Figure 2.2: *The coordinates of the seed in the lower left corner propagates to all pixels in four shader passes. Image extracted from [11].*

2.4 Terrain rendering

Terrains are a common component in computer graphics simulations. One approach to terrain rendering is to use heightmaps. A heightmap can be an image, where the pixel values represent the terrain height at different locations. The heightmap can thus be seen as a function that maps a horizontal coordinate to a vertical coordinate, $f(x, y) = z$.

While being used frequently, terrain rendering is a challenging task. Typically, terrains cover large areas and have small-scale local deformations. This combination requires a large number of triangles which is detrimental to performance in terms of memory usage and computations. However, due to the large area a terrain spans, only a small part of the terrain is visible at any given time. Furthermore, parts of the terrain that are far away can be rendered with less detail than close-up terrain. These two facts have given rise to different types of level-of-detail (LOD) algorithms for efficient terrain rendering [19, pp. 13-14].

2.4.1 Chunk-based level-of-detail

Ulrich [17] presented an approach to rendering huge terrains by dividing the terrain into separate meshes, referred to as chunks. During a preprocessing step, each chunk is created at different LODs. Figure 2.3 shows a terrain being represented by three different LODs. All chunks with their respective LOD are stored in a quadtree. The root of the quadtree consists of a low-detail representation of the entire terrain. The children of each node represent a portion of its parent, at a higher LOD. Besides its mesh, each chunk stores a maximum geometric error, δ. This value represents how much the mesh deviates from the full-detail mesh.

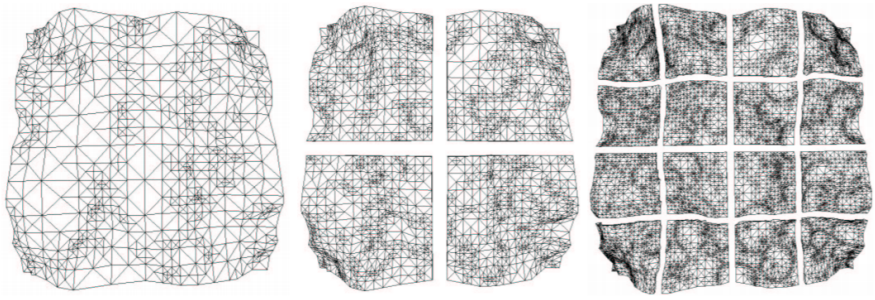


Figure 2.3: *The same terrain at three different level-of-details. Image extracted from [17].*

During rendering, the quadtree is traversed from the root. Given the current node, ρ is calculated according to:

$$\rho = \frac{\delta}{D}K \tag{2.2}$$

where δ is the geometric error, D is the distance from the viewpoint to the closest point on the chunk and K is a perspective scaling factor that accounts for field-of-view and viewport size. If ρ is below a certain threshold, the chunk is rendered, otherwise its four children are recursively examined. This causes terrain geometry far away to be rendered with less detail than close-up geometry.

However, this approach comes with two main issues. The first issue is cracks where neighboring chunks at different LOD meet. The second issue is popping, caused by new vertices being introduced as a terrain chunk is replaced by one with higher LOD. Figure 2.4 illustrates cracks and popping on a terrain.

Figure 2.4: *Two terrain patches rendered at the same LOD (left). Crack and popping introduced when the right-most patch is rendered at a higher LOD (right). The new vertex is shown with a black circle. Figure recreated from [17].*

Cracks can be avoided by adding geometry to one of the meshes that penetrates the other mesh. Popping can be solved by adding a uniform morphing parameter for each chunk, that offsets the vertices in the vertical direction. The morph parameter is linearly interpolated between 0 and 1, such that it is 0 for a chunk that is about to split, and 1 when a chunk begins to merge into a lower LOD. By sampling the height from the parent chunk, the current chunk interpolates its vertices' vertical component. This gives a smooth transition.

As an actual example, Frostbite 2, the game engine that powers Battlefield 3, uses a quadtree approach to terrain rendering. The terrain mesh at the lowest LOD is represented by a rectangular mesh with 33 by 33 vertices. When rendering, each vertex fetches its height from a heightmap and displaces its vertical component [1].

3
Related work

This chapter provides an overview of previous work on dynamic terrain, including deformation of rigid and granular material. Dynamic terrain typically consists of multiple components, including terrain representation, collision detection, deformation representation, applying deformations and handling permanent deformations on large terrains.

3.1 Uniform deformation shape

Frisk [5] developed a system for rendering permanent snow trails after vehicles on large terrains based on Bezier curves.

The approach used a composite cubic Bezier curve to represent trails after vehicles. Points are sampled after the vehicle at given intervals and incorporated to the composite Bezier curve. The shape of the Bezier curve is used to create and continuously update a mesh that represents the trail in the snow.

The author motivated the use of Bezier curves by comparing them to a heightmap based approach which would consume more memory. The larger the terrain, the smaller the area each heightmap pixel covers. This calls for high resolution textures, which consumes lots of memory on large terrains.

An issue with this approach is trails crossing over each other (which happens when the Bezier curve intersects itself). This results in undesirable flickering. A heightmap based approach does not have this issue.

Furthermore, the proposed system is static in terms of only supporting a uniform trail shape. It is not dynamic enough to support general deformations such as

footsteps or a person falling on the ground. Also, there is no support for ground animation.

In Rise of the Tomp Raider [8, Ch. 18], a technique called *deferred deformation* was introduced to create snow deformations in real-time. Besides causing a trail in the snow where the player walks, the snow is elevated around the trail. See Figure 3.1 for an cross-sectional view of a trail.

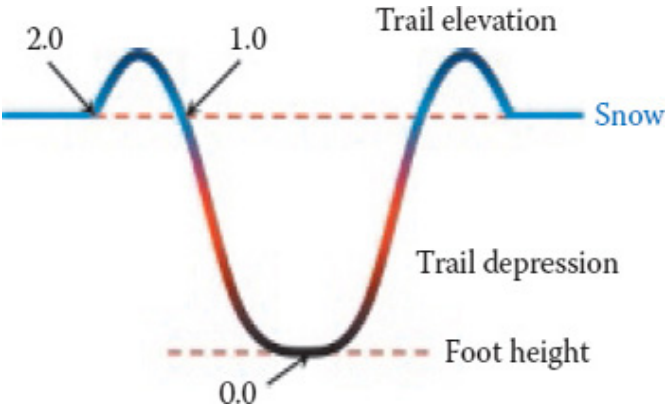


Figure 3.1: *Snow trail shape with a number used to sample different textures at different parts of the trail. Image extracted from [8].*

The systems uses a 1024 x 1024 32-bit texture, called the *deformation heightmap*. When rendering to the deformation heightmap, dynamic objects that can cause deformations are represented as points. These points are stored in a global buffer. A compute shader is invoked for each deformation point and writes to a 32 x 32 pixel area around that deformation point. The value written for a pixel is given by:

$$height + distance^2 \cdot scale \tag{3.1}$$

where $height$ is the height of the deformation point, $distance$ is the horizontal distance to the deformation point from the current pixel and $scale$ is an artistic parameter that changes the shape of the trail. (Higher scale values result in a narrower trail.)

Before writing the new pixel value to the deformation heightmap, a min operation is performed between the new and old pixel value, since there can be multiple deformation points affecting the same pixels.

The value given by Expression 3.1 is written to the 16 most significant bits of the deformation heightmap. The remaining 16 bits are used to store the height of the deformation point.

When rendering the actual snow, the deformation heightmap is sampled and the vertices are vertically offset according to:

$$min(snowHeightmap, deformationHeightmap) \tag{3.2}$$

where $snowHeightmap$ is the base height of the vertex and $deformationHeightmap$ is the sampled height from the deformation heightmap at the current location. However, this is only done if the corresponding deformation point (stored in the 16 least significant bits) is below the snow height. Since it is not known whether the deformation point is above or below the snow cover until rendering, the deformation heightmap is merely a representation of potential deformations, thus the term *deferred* deformation.

The end result is a trail that resembles the shape of a quadratic curve, see Figure 3.1.

To allow for more artistic control, different textures are applied at different stages of the trail. This is achieved by generating a value between 0 and 2 along the shape of the trail and use this number during texture sampling, see Figure 3.1.

This system has some limitations. First, there is no support for animating the snow. This limitation inhibits simulation of granular material. Secondly, the trail is very uniform and follows the shape of a quadratic curve. Each object is represented as a point and the shape of objects is not taken into account. While it would be possible to specify different mathematical formulas for the trail of different shapes, it is probably not dynamic enough to handle arbitrary object shapes.

3.2 Deformation by modifying the terrain mesh

Assassin's Creed III [15] features a snow deformation system that persistently stores deformations from multiple characters over a large terrain.

The snow mesh is a copy of the terrain mesh and offset in the positive Y direction. When a character steps into a new triangle of the snow mesh, the triangle is removed by removing the corresponding indices from the index buffer. By using render-to-vertex-buffer, the removed triangle is replaced by a tessellated triangle, where some triangles are pushed down according to an approximation of the character. Figure 3.2 illustrates this by animating a box over the terrain.

One disadvantage to this approach is the use of a uniform tessellation factor for all replaced triangles. This causes some areas to be over-tessellated, which is bad for performance. Furthermore, a potential disadvantage is the ever-growing mesh size, as more and more triangles of the original mesh gets replaced by higher detailed triangles. Furthermore, there is no support for granular material.

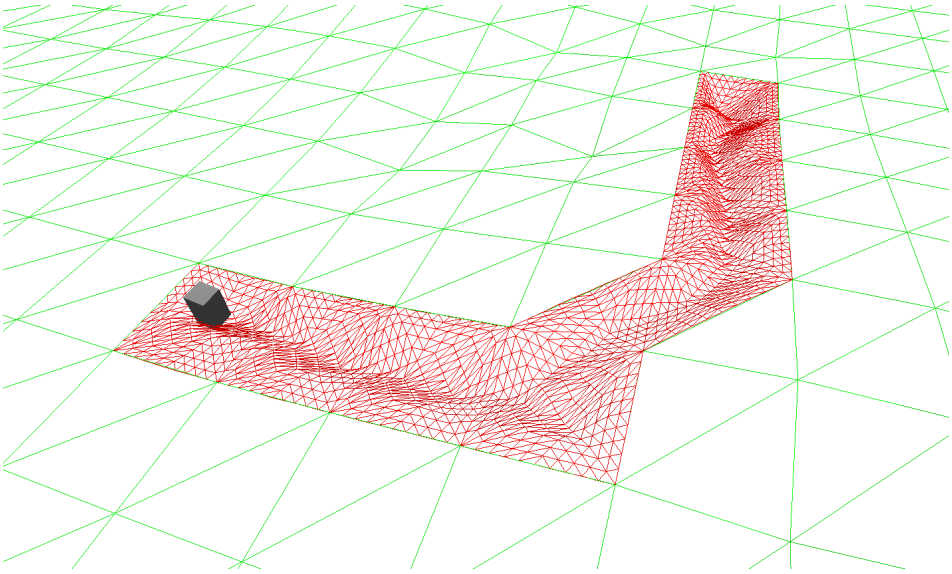


Figure 3.2: *Tessellation and displacement of triangles where the player has walked.*

3.3 Depth buffer based deformation

Aquilio et al. [2] presented a GPU-based algorithm for dynamic terrain simulation that uses the depth buffer to detect intersections. Their algorithm focuses on the interaction between vehicles and terrain.

Initially, the terrain elevation data is stored in a 2D image. A subset of the image data, representing a deformable terrain region, is moved to a GPU buffer. This is achieved through an initial render pass where the camera is positioned beneath the terrain, looking upward. The parameters are such that the viewable region encapsulates the desired subarea of the terrain. The resulting buffer is referred to as the *Dynamically-Displaced Height Map* (DDHM).

Next, objects that interact with the terrain are rendered from below to a texture, with the same view parameters as in the initial step, see Figure 3.3. Fragments are only written to the texture if they are closer to the camera than the corresponding terrain elevation at that point. The result of this texture is used to create an offset map, that represents how much the terrain should be offset at a given point, due to vehicle intersection.

When rendering the terrain to the screen, both the DDHM and the offset map is available to the vertex shader. Both are accessed through a common texture coordinate, and used to offset a vertex in a rectilinear, planar grid.

Finally, the DDHM is updated to be used in the next frame.

This method is efficient in terms of not having to transfer any data between the CPU and GPU. However, it is unfit to model granular materials as the ground is simply removed instead of being animated.

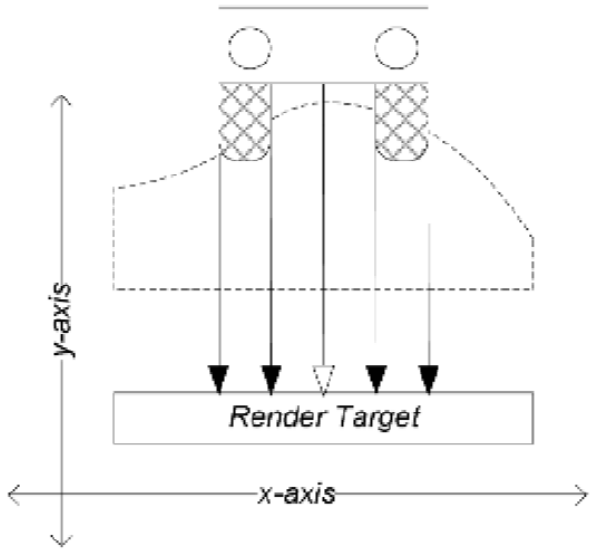


Figure 3.3: *Camera configuration when rendering objects that interact with the terrain. Image extracted from [2].*

A similar project that investigated terrain deformation from vehicle interaction was proposed by Wang et al. [18].

Three textures are used. The first is the *terrain depth texture* which represents the initial and the deformed terrain. The second texture is the *vehicle depth texture*. The third texture is the *depth offset map*. The vehicle depth texture is created by rendering the scene from below, looking up, with a orthographic projection, into a FBO with a depth attachment. The vehicle depth texture is then compared with the terrain depth texture to generate the depth offset map. Finally, by subtracting the depth offset texture from the terrain texture, the new deformed terrain texture is calculated.

The vertices are displaced in the vertex shader by sampling the terrain texture.

Batman: Arkham Origins [3] is a commercial example of using depth buffer based collision detection for ground deformation. The video game contains scenes of buildings covered in snow, where characters walking on the snow produce footsteps dynamically. The approach to dynamically changing the terrain is similar to that of Wang et al. Displacement heightmaps are generated at runtime by rendering all objects that interact with the snow from below, to a texture. A frustum

with the same depth as the height of the snow cover is used. A pixel value of 1 in the displacement map indicates no intersection with the snow, while any value between 0 and 1 would indicate some intersection with the snow. Next, a blur filter is applied to the displacement map and it is combined with the older one. To render the deformation, relief mapping is used on consoles and tessellation on PC.

The shading process uses two materials. One for flat snow (snow that has not been interacted with), and one material for fully flattened snow (snow that has been compressed). In between those areas, the diffuse and specular color of the two materials are linearly interpolated using the values in the displacement map.

Common for all work outlined in this subsection is the use of a depth buffer to modify the terrain heightmap. While achieving real-time performance, there is a lack of realism in how the terrain is modified. The ground is simply subtracted upon intersection. There is no support for modelling granular terrain material that would move around during intersection and collapse if too great slopes build up.

3.4 Deformation of subdivision terrains

Schäfer et al. [13] presented a system for fine-scale real-time deformations on subdivision surfaces that runs entirely on the GPU. The deformable surfaces are represented as displaced Catmull-Clark subdivision surfaces. The displacement is created by moving the control points of quadratic B-splines.

Both low-frequency and high-frequency deformations are supported. Low-frequency deformations, as well as physics simulation and collision detection, runs on the CPU. Changes in the base mesh cause the modified control points to be uploaded to the GPU each frame for consistency.

High-frequency deformations are updated and stored on the GPU in a tile-based memory format. Each tile is stored as a texture and maps to a base face of a subdivision surface. The texture pixels of the tiles are interpreted as control points of a bi-quadratic B-spline, which is used to displace the surface.

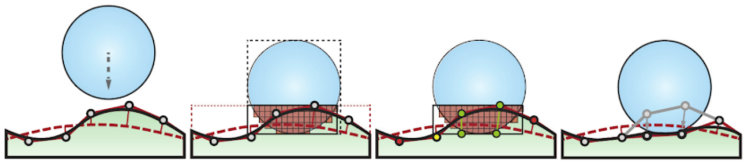


Figure 3.4: *Deformation process. Image extracted from [13].*

Figure 3.4 outlines the algorithm for surface deformation. Each object is approximated with an oriented bounding box, in order to efficiently find colliding objects.

The overlapping geometry is then voxelized, on the GPU. Next, the control points of the deformable surface are moved along the negative normal direction of the base surface, until they no longer intersect the other object. This is achieved by casting a ray from the current control point's world space position. The new position of the control point is stored, and subsequently used to offset the surface during rendering.

The system enables fine-scale deformations over large surfaces with relatively small computational overhead. However, the algorithm lacks support for animating the terrain, which makes it inadequate for rendering granular terrain material.

3.5 Handling deformation on large terrains

Crause et al. [4] presented a general framework for real-time, permanent deformations on large terrains. Deformations are represented as stamps. Three types of stamps are discussed: mathematical, texture based and dynamic. Mathematical stamps apply a mathematical formula over the stamp area. Texture stamps store a heightmap that is used to offset the terrain. Dynamic stamps are changed over time. An example of using dynamic stamps is to produce a shockwave effect, where the stamp is animated to convey the wave propagating.

A stamp is added to the terrain heightmap by combining the two in a shader pass and rendering the new terrain heightmap to an FBO. A texture stamp would take the stamp as a texture input, while a mathematical stamp calculates the height given a mathematical formula in the fragment shader.

After modifying the terrain, the deformation system updates the normal maps of the terrain so that they can be used during rendering. Furthermore, the terrain is streamed back to the CPU to be used in collision detection.

Besides stamp-based deformation, adaptive tessellation and a tile-based LOD scheme is utilized to handle large terrains.

Another tile-based approach to dynamically modify large terrains was presented by Yusov [19, Ch. 2]. The proposed system uses a quadtree based terrain system similar to that of Ulrich [17]. Furthermore, deformations are supported through displacement maps.

The initial heightmap is stored in a quadtree data structure on the GPU by dividing it into different patches and at different resolutions. An efficient GPU accelerated compression and decompression scheme is used to reduce memory consumption. The reconstruction of the compressed quadtree is bounded by a user-defined error tolerance and can be performed in parallel on different patches.

During run time, the GPU maintains a decompressed unbalanced quadtree which represents the current terrain for some given view parameters. When rendering

the terrain, a view-dependent tessellation scheme is used. Skirts[1] are used to hide cracks between neighboring patches.

Dynamic modifications to the terrain are represented with displacement maps. They are applied to the terrain in two parts. First, the displacement map is applied to the current resolution level of the affected patches. This is efficiently performed by render-to-texture. However, if the current patch is too coarse, some modifications might get lost. Therefore, if a modification is applied to patches that are not in the finest resolution level, the finest resolution level is decompressed and updated with the modification. Next, the modified heightmap must be recompressed in a bottom-up fashion until it reaches the currently displayed resolution level. This is performed asynchronously. The compressed heightmaps for patches coarser than the currently displayed are not modified at this stage. Instead, they are updated when they are needed.

3.6 Deformation of granular terrain

Previous methods discussed fall short on modelling granular terrain, such as sand or granular snow.

Animation of granular material has been investigated in many projects. The approaches can broadly be categorized into heightmap-based and particle-based. Typically, particle-based solutions tend to achieve higher physical accuracy, while heightmap-based solutions suit real-time simulations better. There are also hybrid solutions, that use a combination of heightmaps and particles to model the ground during deformation [7].

3.6.1 Heightmap based approaches

Sumner et al. [16] presented a general model for ground deformation of granular material. Figure 3.5 outlines the algorithm. The continues ground is discretized into a heightmap with vertical columns. Each column performs ray casting to detect if a rigid body intersects the column. When a collision is registered, the corresponding columns adjust their height so that the rigid body does not intersect them any longer. Each column calculates the amount of material that will be displaced as $m = \Delta H \cdot a$ where ΔH is the intersection depth and a is the compression ratio. The displaced material is moved to the closest column that did not register a collision.

[1] Additional geometry at patch boundaries.

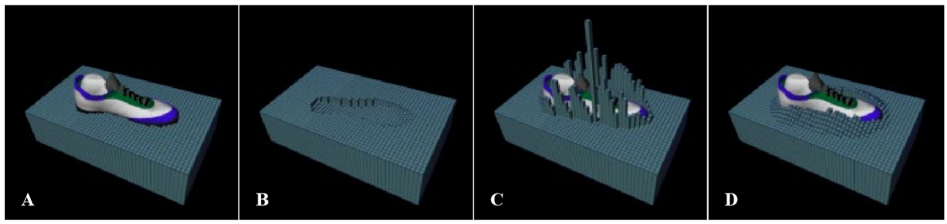


Figure 3.5: *Shows the steps for terrain deformation by displacing material outward and evening out steep slopes. Image extracted from [16].*

Next, a erosion step is performed to even out large height differences among neighboring columns. For a column ij and a neighboring column kl, the slope is calculated according to:

$$s = \frac{tan^{-1}(h_{ij} - h_{kl})}{d} \tag{3.3}$$

where h is the height of the column and d is an artistic parameter. Among all neighbors where s is greater than a specified threshold, the average height difference is calculated according to:

$$\Delta h_{avg} = \frac{\sum(h_{ij} - h_{kl})}{n} \tag{3.4}$$

where n is the number of neighbors with too great slope. The amount of material moved to the n neighbors is given by $\sigma \frac{h_{avg}}{n}$, where σ is a fractional constant that affects the roughness of the material.

By changing the threshold for when material is moved between neighboring columns, how many erosion steps are performed in each frame, the compression ratio and the roughness parameter, different materials such as snow, sand and mud can be simulated.

The resolution of the heightmap dictates the quality of the simulation. Higher resolution results in a finer simulation, but at the expense of greater memory and computational load. As an optimization step, a bounding box of rigid bodies are projected onto the terrain to find areas where deformation could occur. Therefore, the algorithm for collision detection, material displacement and erosion is not performed on the entire grid.

While producing dynamic deformations, the model has some limitations. One is the uniform distribution of displaced material. A more realistic approach would take the velocity of a rigid body into account to move more material in the direction of travel. Furthermore, the model does not remember previous deformations. A more accurate model would take precious compression of material into

account.

Onoue and Nishita [9] builds on top of the work by Sumner et al. One of the main contributions is an algorithm that allows granular material to be put on top of intersecting objects. This is achieved by using a Height Span Map (HS Map) to represent objects interacting with the ground and material on top of the objects.

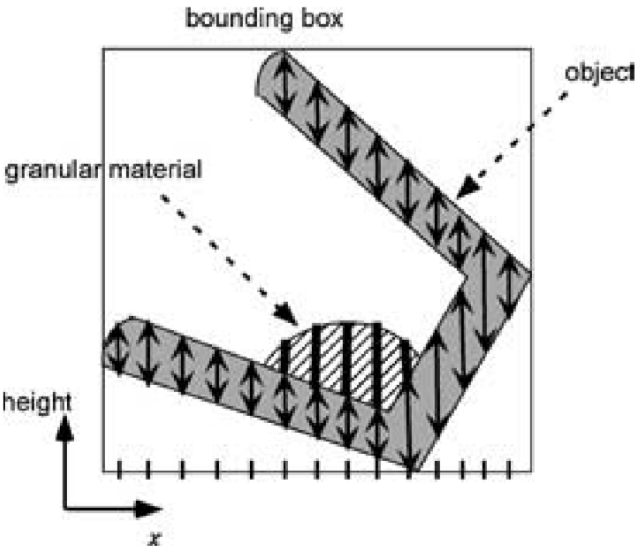


Figure 3.6: *Cross sectional view of a HS Map for a bucket that contains some granular material. Image extracted from [9].*

A HS Map is constructed for each object by rendering each polygon of the object from below. The rendering parameters are such that the resulting rasterization has the same resolution as the heightmap that represents the terrain. Each pixel of the HS Map is allocated a list structure that stores tuples of (d, n), where d is the current pixel depth and n indicates whether the surface direction is inward or outward. After all polygons have been rasterized, the height spans are generated by pairing data points where one has n = "outward" and the other has n = "inward". If the object contains granular material, the height of the material is found by rasterizing the material's polygons, represented as bars in Figure 3.6.

A bounding box intersection test is performed between objects and the terrain. If a collision is detected, the HS Map is updated to ensure that each column in the HS Map aligns with a column in the heightmap. Next, using the height spans, a narrow collision detection is performed (between columns of the terrain heightmap and the corresponding height span of the object).

Next, material is moved to the boundary between the object and the terrain. If the object is falling down, the material is moved in a similar fashion to that of

Sumner et al. However, for objects being dragged horizontally, the direction of the objects is taken into account. A column ij moves material to a neighboring column kl if $\mathbf{d}_{\mathbf{object}} \cdot \mathbf{d}_{\mathbf{column}} \geq 0$, where $\mathbf{d}_{\mathbf{object}}$ is the horizontal direction of the object and $\mathbf{d}_{\mathbf{column}} = (k, l) - (i, j)$.

After the material has been moved, an erosion step similar to that used by Sumner et al. evens out areas with too steep slopes (at object boundaries). The erosion step allows material to move from the ground heightmap to height spans and vice versa.

During rendering, the material on top of objects is rendered by connecting the height values of each height span to a polygonal mesh.

3.6.2 Particle based approaches

Rungjiratananon et al. [12] used the Smoothed Particle Hydrodynamics (SPM) method and the Discrete Element Method (DEM) to model the interaction between granular sand and fluid. The sand is modelled using DEM and the fluid is modelled using SPM. On a limited number of particles, the authors achieved interactable frame rates. However, the system is not performant enough to be used on a large terrain.

Heyn [6] developed an offline system for simulated vehicle tracks on granular terrain using particles. Rigid bodies interacting with the terrain, and the terrain itself, are represented by a union of spheres through a spherical decomposition process. Collision detection among the spheres and the dynamics of the spheres are efficiently computed in parallel on the GPU. While achieving high-detail deformations, this method is not suitable for real-time simulations.

3.6.3 Hybrid approaches

Holz et al. [7] presented a hybrid approach to soil simulation. The focus was on achieving realistic behavior in real-time to build a system that could be used in areas such as virtual reality training simulators for bulldozers and planetary rovers.

The hybrid approach was motivated by the more realistic behavior of particle simulations and the higher performance of grid simulations.

Collisions with rigid bodies are detected by doing ray casting from terrain vertices that are below a bounding box of a potentially colliding object. Upon collision, the heights of the corresponding columns in the soil grid are reduced and replaced by spherical rigid body particles. The particles interacting with each other are simulated using a physics engine. When a soil particle reaches a state of equilibrium, it is removed from the simulation and the height of the corresponding column in the soil grid is increased to match the particle's volume.

Besides using particles to simulate the soil moving around, the soil grid can move material among neighboring columns directly as an optimization. This is done by having each column exchange material between its top and right neighbor. This

ensures that the entire grid is updated, without having to examine each neighbor for every column.

4

Method and implementation

This chapter outlines the algorithm and the implementation of the proposed system for volume-preserving terrain deformations in real-time.

4.1 Terrain deformation algorithm

The terrain deformation algorithm can roughly be divided into four steps. These are detection collision between objects and the terrain, displacing the intersected terrain material to the edge of the intersecting object, evening out steep slopes and finally rendering the deformed terrain.

4.1.1 Terrain representation

The terrain is stored on the GPU in a texture with one 32-bit unsigned integer channel, referred to as the *heightmap*. The resolution of the texture is defined by a user parameter. To achieve a higher precision than integers, each integer is stored 10 000 times greater than the rendered height. Thus, during rendering, each integer value is divided by 10 000 to get the correct height. This number is referred to as the *heightmapScale*. The use of integers is motivated by their ability to represent numbers exactly, compared to floating-point number systems. This is further explained and motivated in Section 4.3.

4.1.2 Detecting collisions between objects and the terrain

Collisions between objects and the terrain are detected by using a depth buffer-based approach similar to that of [2], [18] and [3]. All objects that can interact with the terrain are rendered from below to an FBO with a depth attachment texture. This texture is referred to as the *depth texture*. An orthogonal frustum is

used, with a width, height and depth such that it covers the entire terrain. Figure 4.1 shows the view and projection setup when rendering the objects.

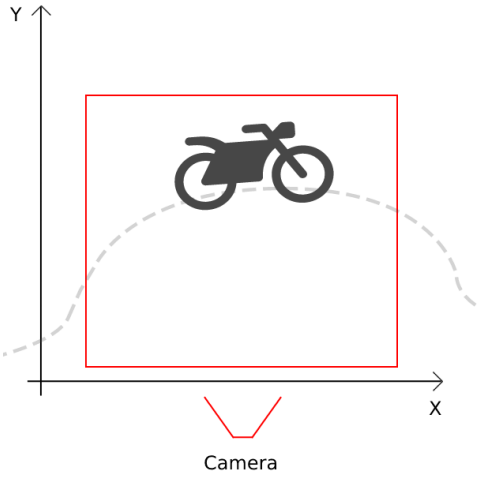


Figure 4.1: *Projection and view parameters when rendering the depth texture. The terrain is shown as dashes for visualization purposes, but is not rendered to the depth texture.*

4.1.3 Calculating the penetration texture

This subsection introduces the *penetration texture*. The penetration texture uses four channels of 32-bit integers to store information about the terrain and objects in the terrain, which is necessary to calculate how material should be displaced.

The penetration texture is calculated in a shader program that takes the heightmap and the depth texture as input. There are three different types of pixels in the penetration texture. The first type is *obstacles*. A pixel is considered an obstacle if it is not penetrating the terrain, but its depth value is just above the terrain. This is used to avoid displacing material to columns occupied by objects. The second pixel type are the columns that penetrate the terrain. The third type are the *seeds*. A pixel is considered a seed if it is not penetrating the terrain and if it is not an obstacle.

Table 4.1 shows an overview of the content in the penetration texture. The red and green channels are set to the coordinates of seed columns and 0 for all other columns. The pixel type is store in the blue channel and the alpha channel stores the penetration of the terrain at that point.

Table 4.1: *Values for the different channels in the penetration texture.*

Channel	Value
r	X coordinate of seeds (otherwise 0)
g	Y coordinate of seeds (otherwise 0)
b	Pixel type (seeds = -3, obstacles = -2, all remaining = -1)
a	Penetration value

Calculating the penetration value

The penetration value is calculated by comparing the value in the depth texture with the corresponding value in the heightmap. Since the depth texture values are within the range $[0, 1]$, they must be converted to the corresponding terrain heights. This is done by scaling the values by the height of the frustum and by heightmapScale. Next, the minimum of each heightmap value and its corresponding transformed depth value is computed and subtracted from the heightmap value. The resulting value is the penetration value within the range $[0, frustumHeight \cdot heightmapScale]$. A value of 0 indicate no penetration, while a positive value indicates how much material the column should displace. See Listing 4.1 for the corresponding GLSL code.

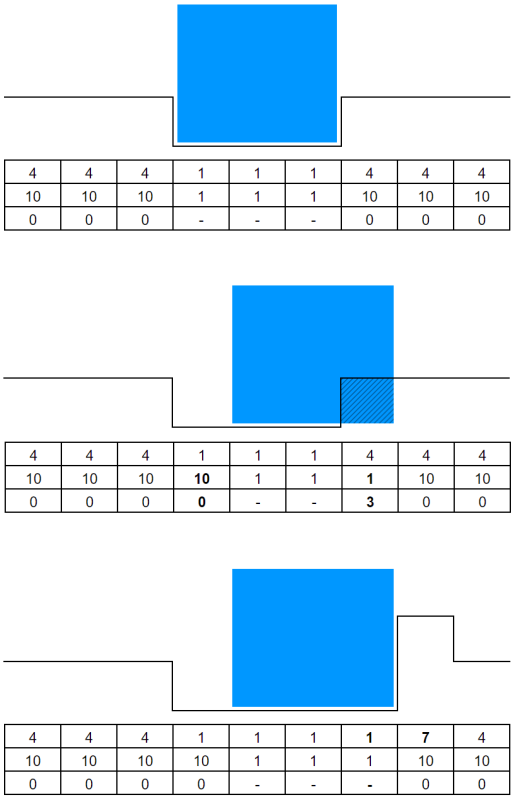
```
1 uint depthValue = uint(texture(depthTexture, texCoord).r * frustumHeight↩
      * heightmapScale);
2 uint heightmapValue = texture(heightmap, texCoord).r;
3 uint penetration = heightmapValue - min(heightmapValue, depthValue);
```

Listing 4.1: *Calculation of penetration value.*

Methods such as [2], [18] and [3] use the penetration value to directly alter the terrain. However, that results in the terrain material being subtracted from the terrain. Instead, as explained in the upcoming sections, this method displaces the material of the terrain in a volume-preserving way.

Deformation scenario

The use of obstacle columns in the penetration texture addresses a shortcoming in [16] of uniformly displacing material in all directions. By setting the columns that doesn't intersect the terrain, but are occupied by objects, to obstacles, it is possible to treat them differently when material is displaced. Figure 4.2 shows a scenario in which a box is moved through the terrain whereafter the terrain deforms. The use of obstacles ensures that the material moves in the same direction as the object. The corresponding heightmap, depth texture and penetration texture are visualized. The next section explains how the actual material displacement is performed.



4	4	4	1	1	1	4	4	4	Terrain heightmap
10	10	10	1	1	1	10	10	10	Depth texture (scaled)
0	0	0	-	-	-	0	0	0	Penetration texture

4	4	4	1	1	1	4	4	4	Terrain heightmap
10	10	10	**10**	1	1	**1**	10	10	Depth texture (scaled)
0	0	0	**0**	-	-	**3**	0	0	Penetration texture

4	4	4	1	1	1	**1**	**7**	4	Terrain heightmap
10	10	10	10	1	1	1	10	10	Depth texture (scaled)
0	0	0	0	-	-	**-**	0	0	Penetration texture

Figure 4.2: *Cross-sectional example of a blue box deforming the terrain in three steps. The values in the depth texture have been scaled to match the values in the heightmap. Dashes in the penetration texture represent obstacles.*

4.1.4 Displacing intersected material

Once the penetration texture has been calculated, the intersected material is displaced to the closest column in the heightmap that is not intersected. To do this, each intersected column needs to know which column in should displace its material to. This is achieved by computing the distance transform of the penetration texture. The resulting texture is referred to as the *distance texture*.

The distance transform is calculated using Jump Flooding. The algorithm is implemented in a shader program that initially takes the penetration texture as input and then iteratively computes the distance texture in a ping-pong fashion. Each iteration updates the red and green channels to store the coordinates of the closest seed. Figure 4.3 shows an example of a distance transform using the Manhattan distance as metric.

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	5	5	5	5	5	5	0	0
0	0	5	5	5	5	5	5	0	0
0	0	5	5	5	5	5	5	0	0
0	0	5	5	5	5	5	5	0	0
0	0	5	5	5	5	5	5	0	0
0	0	5	5	5	5	5	5	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	0	0
0	0	1	2	2	2	2	1	0	0
0	0	1	2	3	3	2	1	0	0
0	0	1	2	3	3	2	1	0	0
0	0	1	2	2	2	2	1	0	0
0	0	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 4.3: *Penetration value of the penetration texture (left). Corresponding distance transform using Manhattan distance (right).*

As explained in the theory section, each iteration of the Jump Flooding algorithm consists of examining eight neighbors for each pixel. If the same order is used when examining neighbors, some directions can get favored over others if multiple neighbors have the same distance. Therefore, the examination order is randomized by feeding the current texture coordinate to a function that generates a pseudorandom value.

Using the distance texture, each column knows the distance and coordinates to the closest column that can receive material. This information is used to displace the material to the contour of intersecting objects. Three different methods are implemented for this. The first is a parallelized version of the algorithm in [16] that requires multiple render passes. The second method uses a SSBO to allow arbitrary writes which enables it to transfer all material in one render pass. The third method works similar to the second, but instead of displacing all material to one column, it tries to distribute the material over multiple columns to avoid high contours around objects.

Method 1: Iterative material displacment algorithm

The iterative method for displacing material works very similar to that of [16]. But instead of being implemented on the CPU, all computations are performed in parallel on the GPU.

First, a shader program takes the distance texture as input and computes how many neighbors of each pixel have a lower distance to the closest seed than the current pixel. This is written to the red channel of the texture. The green channel stores the offset that will be used to update the actual heightmap later. The offset is set using a compression parameter in the range $[0, 1]$. A high value means that more material is compressed and less material is displaced. The blue channel stores the contour distance and the alpha channel stores the penetration value. Table 4.2 shows an overview of the output texture of this program.

Table 4.2: *Values for the different channels.*

Channel	Value
r	Number of neighbors with lower contour distance
g	Offset
b	Distance to closest seed
a	Penetration value

Next, a shader program that iteratively moves material to neighbors with a lower distance value is invoked in a ping-pong fashion. The program takes the texture from the previous shader as input. For each neighbor j with a higher distance value, the program divides the penetration of j with the number of neighbors that will receive material from j. This value is accumulated for each neighbor and written as the new penetration value minus how much the current pixel will transfer to its neighbors.

Since the amount of penetration stored by a column might not be evenly divisible by the number of neighbors that will receive material, not all material might be transferred. Therefore, the modules operation is used to ensure that the terrain is volume-preserving. See the following listing.

```
1 newPenetration = totalReceivedFromNeighbors + oldPenetration % ↩
      numReceivingNeighbors
```

Listing 4.2: *Calculation of updated penetration value.*

The output texture is referred to as the *offset texture* and has the same channel setup as shown in Table 4.2. The offset value (green channel) is updated in a similar way as the penetration value.

A disadvantage to this approach is that it is not trivial to calculate the number of iterations required to move all material to the contour of the objects. Each iteration moves material to the closest neighbors. If a square area of 16 pixels are penetrating the terrain, at least three iterations are needed. The number of iterations needed grows with the size of the biggest penetrating area and the resolution of the heightmap. To find the number of iterations required it would be possible to implement a shader program that finds the maximum distance value in the distance texture. However, in this project, a static value for the number of iterations is used an specified as a user-parameter.

Method 2: SSBO based material displacement algorithm

To avoid the drawback of the iterative method, a second method for displacing material to the contour is implemented. The idea is to use a memory buffer that allows reading and writing to arbitrary locations. In this way, each column can directly transfer its material to the contour.

When the main program first starts, a SSBO is allocated on the GPU and filled with zeros. The size of the buffer is set to $heightmapSize \cdot heightmapSize \cdot 4$, where 4 is the byte size of an integer.

To displace material, a shader program reads from the distance texture to find the coordinates of the closest non-penetrating column and how much material it should receive. The coordinates are converted to an index in the SSBO according to the following listing.

```
int index = texCoord.y * heightmapSize * heightmapSize + texCoord.x * heightmapSize;
```

Listing 4.3: *Conversion from texture coordinate to SSBO index.*

Next, using atomic additions, the SSBO is updated to reflect the material removed from the current column and added to the closest seed. The current heightmap value is also added to the current column. See the following listing.

```
    atomicAdd(data[indexMe], heightmapValue - penetration);
    atomicAdd(data[indexCloest], uint(penetration * (1 - compression)));
```

Listing 4.4: *Update procedure of the SSBO.*

The last step consists of transferring the SSBO data back to the heightmap. However, this is performed after the next step of the algorithm.

The major benefit to this method over the iterative approach is that all material can be moved in a single render pass. This also means that it is not necessary to calculate how many iterations are needed. However, the drawback to this approach is the cost of performing arbitrary atomic writes in the buffer.

Method 3: Direct material displacement with multiple targets

A slightly different version of the SSBO based material displacement algorithm is implemented. Instead of displacing all material to one column, the material is distributed to multiple columns, by iteratively marching in the direction to the closest column. This is referred to as the SSBO based method with multiple targets. The following listing of the shader program shows the approach.

```
    int numTargets = int(ceil(penetration / 1000.0));
    penetration = penetration / numTargets;
    for (int i = 0; i < numTargets; i++) {
        vec2 target = cloestSeedIntCoordinate + dirToClosestSeed * (i + 1);
        int targetSSBOIndex = intCoordinateToSSBOIndex(target);
        atomicAdd(data[targetSSBOIndex], uint(penetration * (1 - compression)));
    }
```

Listing 4.5: *Procedure for displacing material to multiple target columns.*

The motivation behind this method is to avoid too high contours around intersection objects. Instead, material is moved to multiple columns. The number of receiving columns depends on the penetration value. A deeper penetration results in more columns receiving material.

4.1.5 Evening out steep slopes

The result of the previous step is a terrain with unrealistically high edges near objects that previously intersected the terrain. The height of the contours depends on many factors such as the frame time, the resolution of the heightmap (each pixel represents a smaller area in a higher resolution heightmap) and the amount of material to move.

The final step of the deformation algorithm consists of evening out steep slopes to produce a more realistic looking terrain. As in the previous step, different methods are implemented. The first method is based on [16]. It requires two shader programs that runs in an interleaving pattern multiple times each frame. The other method uses an SSBO to allow arbitrary write locations.

Method 1: Two interleaved shader programs

The first shader program writes to a four channel 32-bit unsigned integer texture. Table 4.3 shows the content of the different channels. The main task of the program is to calculate how much material each column should remove and how much material its neighbors should receive.

Table 4.3: *Values for the different channels in the output texture.*

Channel	Value
r	Amount of material to remove
g	Amount of material to move to each receiving neighbor
b	Column height
a	Obstacle height

To determine which neighbors will receive material, the slope between the current column and its eight neighbors is calculated according to:

$$s = \frac{tan^{-1}(h_{ij} - h_{kl})}{d} \tag{4.1}$$

where h_{ij} is the current column's height, h_{kl} is the neighbor's height and d is an artistic parameter.

For those n neighbors with a slope less than a user-defined threshold, referred to as the *slopeThreshold*, the average height difference is calculated according to:

$$h_{avg} = \frac{\sum(h_{ij} - h_{kl})}{n} \tag{4.2}$$

Next, the average height difference h_{avg} is divided by the amount of receiving neighbors and multiplied by a *roughness* parameter. The result is written to the green channel of the texture. Both Equation 4.1 and 4.2 are borrowed from [16].

The second program takes the texture from the first program as input and performs the actual updates to the terrain. Each column removes the material stored in the red channel, and adds the material stored in the green channel of its eight neighbors. The shader writes to a texture with two channels, each storing a 32-bit integer. The red channel stores the new terrain height, while the green channel stores the obstacle height.

These two shaders are interleaved and runs a user-defined number of times. In [16], a threshold of the terrain slopes was used to decide when the process should stop each frame. However, as the heightmap data is stored on the GPU and the shader invocations are controlled from the CPU this is not as trivial to do.

Finally, a third shader program reads from the new two-channel heightmap and writes the values to the main one-channel heightmap.

Method 2: SSBO based approach

The SSBO based method performs the same calculations as the previous method, but does it in one shader program. This is possible because the SSBO allows each column to write data to the other columns, and can thus move its material directly. This is differently from the other method, which first calculates how much each neighbor should receive, and then has each neighbor read that value in a second shader program.

The SSBO based method reuses the same SSBO from the material displacement step. As a finishing step, a shader reads the SSBO and writes to the heightmap to produce the new heightmap.

The main advantage to this approach is that it only requires one shader to be repeatedly called. This saved the state change overhead that comes with the previous method. However, the arbitrary writes produce some overhead.

4.1.6 Rendering the terrain

After the new heightmap has been calculated through the previous steps, the normal map is updated in a shader program. This is done by calculating the symmetric derivative for each pixel according to:

$$f'(x) = \frac{f(x+1) - f(x-1)}{2} \tag{4.3}$$

Depending on the desired look, a user-configurable parameter decides how many times to apply a 3 by 3 Gaussian kernel to blur the normal map. This is achieved using a low-pass filter and causes the ground to look more smooth.

To enable hardware linear filtering of the heightmap, a render pass moves the updated columns of the integer heightmap to a floating-point texture.

When rendering the terrain, a rectilinear grid is used. The grid is tessellated further in a Tessellation Control Shader (TCS) based on the distance from the viewpoint to the center of each edge. In the Tessellation Evaluation Shader (TES), the vertices are interpolated and their Y component is offset according to the values in the heightmap.

4.2 User parameters

Multiple user parameters can be used to change the appearance and performance of the deformation system. The program accepts a settings file where each parameter can be set. Table 4.4 shows an overview of the parameters.

Table 4.4: *Overview of user parameters.*

Parameter	**Description**
terrainSize	Scales the terrain during rendering.
numVerticesPerRow	Sets the resolution of the terrain mesh.
heightmapSize	Sets the resolution of the terrain heightmap.
compression	The ratio of compressed versus displaced material in the range $[0, 1]$. A value greater than 0 makes the system non volume-preserving.
roughness	Factor in the range $[0, 1]$ that scales the amount of material to displace when evening out steep slopes. Lower values cause a smoother terrain.
slopeThreshold	Threshold used to decide whether two columns should even out their slope.
numIterationsDisplaceMaterialToContour	The number of iterations to run the shader for displacing material iteratively to the contour.
numIterationsEvenOutSlopes	The number of iterations to run the shader program(s) for evening out large column height differences.
numIterationsBlurNormals	The number of times to blur the normal map. A higher number produces smoother normals and a smoother looking terrain.

4.3 Texture formats

All values that represent column heights (such as the heightmap and the penetration texture) are stored as integers. More precisely, the heightmap is stored as a one channel 32-bit unsigned integer. When rendering the heightmap, the pixel values are divided by 10 000. This allows decimal precision. A 32-bit unsigned integer can store values from 0 to 4 294 967 295. Dividing it by 10 000 allows a terrain height to vary from 0 to 429 496 with a precision of 0.0001.

The motivation behind integer-based textures is to ensure a volume-preserving system. Initially, floating-point textures were used to represent column heights. However, this caused material to either disappear or appear out of thin air occasionally when a height value was updated (such as during displacing material or evening out steep slopes). This problem was caused by the limited precision of floating-point number systems. Whenever a column stores a new floating-point number, it might be rounded to the closest representation if it cannot be stored exactly. For instance, imagine a column that stores the floating-point value A and another column that stores the value B. If the second column should displace terrain material to the first column, the first column would store $A + B$. However, this number might not be able to be stored exactly. This causes material to either disappear or be added.

4.4 Memory usage

The system makes use of multiple textures. To limit memory usage and to make sampling faster, different texture formats are used depending on the needs. Table 4.5 lists the textures used by the SSBO based methods for displacing material and evening out steep slopes (commonly referred to as the SSBO based approach). Table 4.6 lists the textures used by the iterative displacement method and the interleaving method for evening out steep slopes (commonly referred to as the iterative approach).

Table 4.5: *Textures and their formats for SSBO based approach. (2) indicates that two textures are used due to ping-ponging.*

Name	Channels per pixel	Channel data type
Heightmap	1	32-bit unsigned integer
Heightmap for rendering	1	32-bit float
Normalmap	4	32-bit float
Blurred normalmap (2)	4	32-bit float
Depth texture	1	16-bit float
Penetration texture	4	32-bit signed integer
Distance texture (2)	4	32-bit signed integer

Table 4.6: *Textures and their formats for the iterative approach. (2) indicates that two textures are used due to ping-ponging.*

Name	Channels per pixel	Channel data type
Heightmap texture	1	32-bit unsigned integer
Heightmap for rendering	1	32-bit float
Normalmap	4	32-bit float
Blurred normalmap (2)	4	32-bit float
Depth texture	1	16-bit float
Penetration texture	4	32-bit signed integer
Distance texture (2)	4	32-bit signed integer
Distance texture with neighbor data	4	32-bit signed integer
Offset texture (2)	4	32-bit signed integer
Packed texture for evening out slopes	2	32-bit unsigned integer
Average height difference texture	4	32-bit unsigned integer
Evened heights texture	2	32-bit unsigned integer

The memory usage for the SSBO based approach is $78 \cdot heightmapSize^2$ bytes. This includes the memory used by the SSBO. A heightmap size of 1024 would require 81 788 928 bytes of memory (82 MB) for the deformation system.

The memory usage for the iterative approach is $126 \cdot heightmapSize^2$ bytes. A heightmap size of 1024 would require 132 120 576 bytes of memory (132 MB) for the deformation system.

4.5 Optimizations

On a large terrain with a high resolution heightmap, the system performs very poorly. Multiple optimization techniques are employed to improve the performance.

A major optimization on large terrains is achieved by only updating the terrain on areas where objects that can intersect the terrain currently are. This saves a lot of processing, since it is usually relatively few columns that are updated each frame. To achieve this, each object calculates a pixel area to be simulated by using its position and size. These areas are referred to as *active areas*. When invoking the shader programs for the deformation system, the viewport and the translation and scale matrices are set in such a way as only a subset of the pixels are invoked by fragment shaders. This results in fewer pixels being simulated, while the number of render calls increases (as a function of the number of objects that can affect the terrain).

Another optimization is combing data to the same texture to avoid having to sample from multiple textures.

4.6 Implementation details

The system is implemented in C++ with OpenGL 4.3. To access OpenGL functionality, GLFW and GLEW is used. GLFW is a library that facilitates creating a window and obtaining an OpenGL context. GLFW is also used to handle input events from mouse and keyboard. GLEW is an extension loading library that is used to load pointers to modern OpenGL functions.

Several other libraries are included in the project. These are TinyObjLoader, LodePNG and GLM. TinyObjLoader is used to load OBJ files, LodePNG is used to decode PNG files and GLM is used to facilitate vector and matrix operations.

5
Result

This chapter provides an evaluation of the proposed system. More specifically, different distance metrics are tested for the distance transform in terms of the visual result. The performance of all individual steps of the algorithm is measured and presented for different parameters. The performance gain of using active areas is tested. A proof of volume preservation is given. Finally, some visual results of vehicle trails and foot trails are presented.

In the method chapter, three algorithms for displacing material to the contour and two algorithms for evening out steep slopes were introduced. Table 5.1 shows which of these configurations are used when evaluating the system and how they are referred to.

Table 5.1: *Friendly names for different algorithm combinations.*

Friendly name	**Method for displacing material**	**Method for evening out slopes material**
Iterative	Method 1	Method 1
SSBO	Method 2	Method 2
SSBO with multiple targets	Method 3	Method 2

5.1 Evaluation method

All measurements of FPS and time consumption in the evaluation is computed using *glfwGetTime()*. This function is included in GLFW and returns a timestamp in seconds, measured since GLFW was initialized. To measure the time required

for a block of code, a timestamp before and after the code is registered. However, when issuing OpenGL commands, the commands are buffered on the GPU and executed in chunks. This means that when the second timestamp is registered, the GPU might still not have performed any calculations. To work around this, *glFinish()* is used. This function ensures that OpenGL completes all buffered commands before the control is returned to the main program. *glFinish()* is never used in measurements of FPS, due to the overhead induced from waiting for the GPU to finish.

Table 5.2 presents the specification of the computer system used when performing the measurements.

Table 5.2: *System specification when evaluating the system.*

Component	Value
CPU	Intel i5 4400k
GPU	NVIDIA GeForce GTX 660
VRAM	2 GB
RAM	16 GB
OS	Windows 10

5.2 Displacing intersected material

Figure 5.1, 5.2 and 5.3 show the result of displacing material to the contour of intersecting objects using the Euclidean, Manhattan and Chebyshev distance metric for the iterative, SSBO based and SSBO based with multiple targets approach respectively. In this example, a box is penetrating the terrain at an angle of 45 degrees to the heightmap axes. Each heightmap pixel in the terrain is rendered as a vertical bar and colored based on its heights.

The iterative approach distributes material evenly to its neighbors with a lower distance to the contour. This is the reason behind the smooth contour shapes, compared to the SSBO based approach. When using the SSBO based approach, each column moves all its material to one specific column. This makes the contour less smooth.

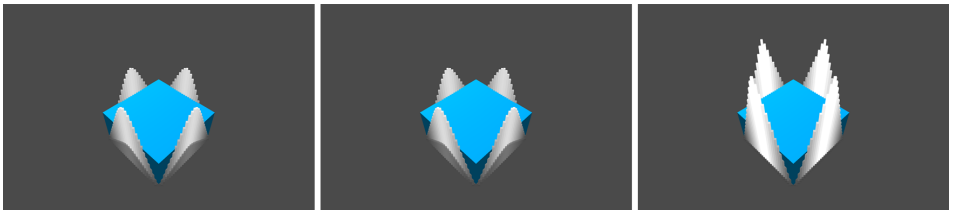


Figure 5.1: *Iterative approach. Euclidean distance (left). Manhattan distance (middle). Chebyshev distance (right).*

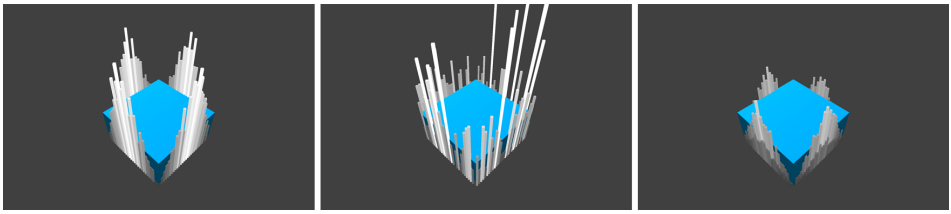


Figure 5.2: *SSBO approach. Euclidean distance (left). Manhattan distance (middle). Chebyshev distance (right).*

Figure 5.3: *SSBO approach with multiple targets. Euclidean distance (left). Manhattan distance (middle). Chebyshev distance (right).*

5.3 Performance of sub steps

This section presents an evaluation of the performance of the different steps in the algorithm. Table 5.3 shows the configuration used, unless otherwise specified. The scene setup consists of a car that is driven over a terrain covered in snow. There is one active area that represents the pixels under the car plus some margin. This means that only a subset of the pixels in the heightmap texture are simulated. The active area corresponds to 3774, 15 100, 60 398 and 241 591 pixels on a heightmap size of 512, 1024, 2048 and 4096 respectively.

Table 5.3: *Configurations for performance evaluation.*

Paramater	Value
terrainSize	200
heightmapSize	4096
numVerticesPerRow	64
numIterationsBlurNormals	1
numIterationsEvenOutSlopes	10
numIterationsDisplaceMaterialToContour	10

Figure 5.4 and Figure 5.5 shows the time in milliseconds for each step of the algorithm when varying the heightmap size for the iterative and SSBO based approach respectively. Each measurement is the average of multiple measurements

from different frames during a 30 second long simulation. Table 5.4 and Table 5.5 shows the same information but in tabular form.

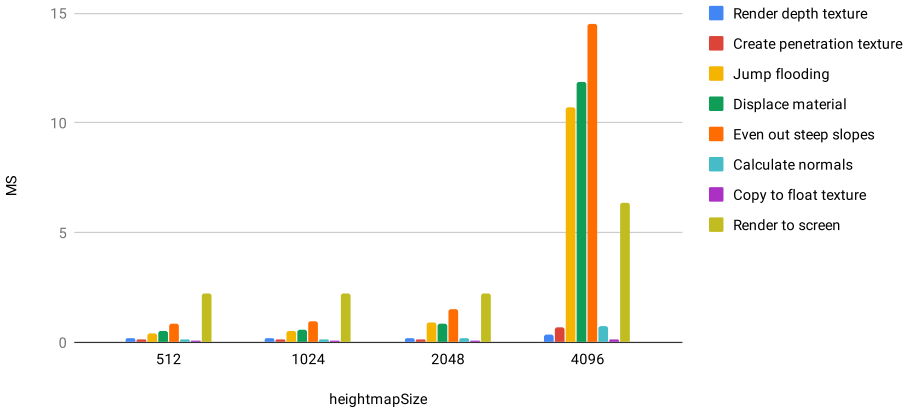


Figure 5.4: *Time measurements for different steps in the algorithm with the iterative approach.*

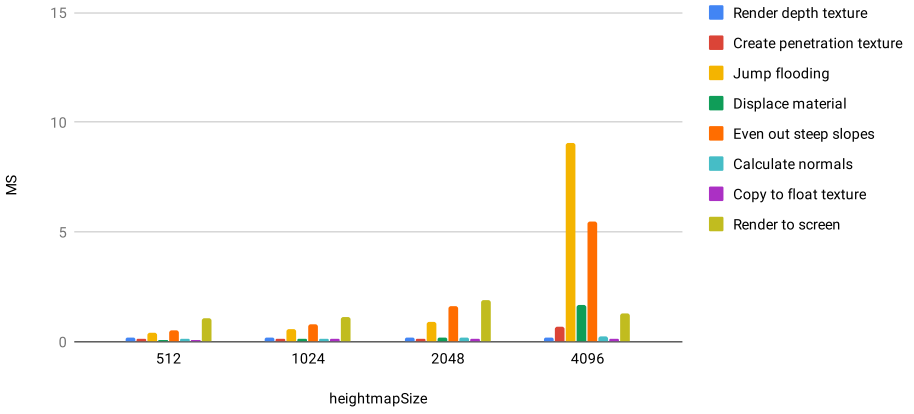


Figure 5.5: *Time measurements for different steps in the algorithm with the SSBO approach.*

Table 5.4: *Time measurements for different steps in the algorithm with the iterative approach. All measurements are shown in milliseconds.*

	heightmapSize			
Step	512	1024	2048	4096
Render depth texture	0.203	0.202	0.201	0.373
Create penetration texture	0.116	0.112	0.118	0.671
Jump flooding	0.403	0.529	0.909	10.692
Displace material	0.513	0.579	0.847	11.864
Even out steep slopes	0.826	0.988	1.531	14.480
Calculate normals	0.148	0.155	0.171	0.760
Copy to float texture	0.102	0.103	0.107	0.127
Render to screen	2.245	2.251	2.207	6.363

Table 5.5: *Time measurements for different steps in the algorithm with the SSBO approach. All measurements are shown in milliseconds.*

	heightmapSize			
Step	512	1024	2048	4096
Render depth texture	0.204	0.207	0.201	0.210
Create penetration texture	0.111	0.116	0.120	0.670
Jump flooding	0.408	0.549	0.905	9.045
Displace material	0.107	0.127	0.177	1.659
Even out steep slopes	0.544	0.773	1.617	5.486
Calculate normals	0.147	0.161	0.175	0.266
Copy to float texture	0.100	0.108	0.108	0.125
Render to screen	1.095	1.107	1.885	1.314

Figure 5.6 shows the performance of the three different algorithms for displacing intersected material. A texture size of 4096 is used and the value for numIterationsDisplaceMaterialToContour is varied. Table 5.6 shows the same data in tabular form, when numIterationsDisplaceMaterialToContour is set to 1.

Figure 5.7 shows the performance of the iterative approach and the SSBO approach for evening out steep slopes on a texture size of 4096. The value for numIterationsEvenOutSlopes is varied.

Table 5.6: *Time in milliseconds for the material displacement step. The iterative approach runs one iteration per frame.*

Method	**Time (ms)**
Iterative approach	2.84
SSBO	1.04
SSBO with multiple targets	1.15

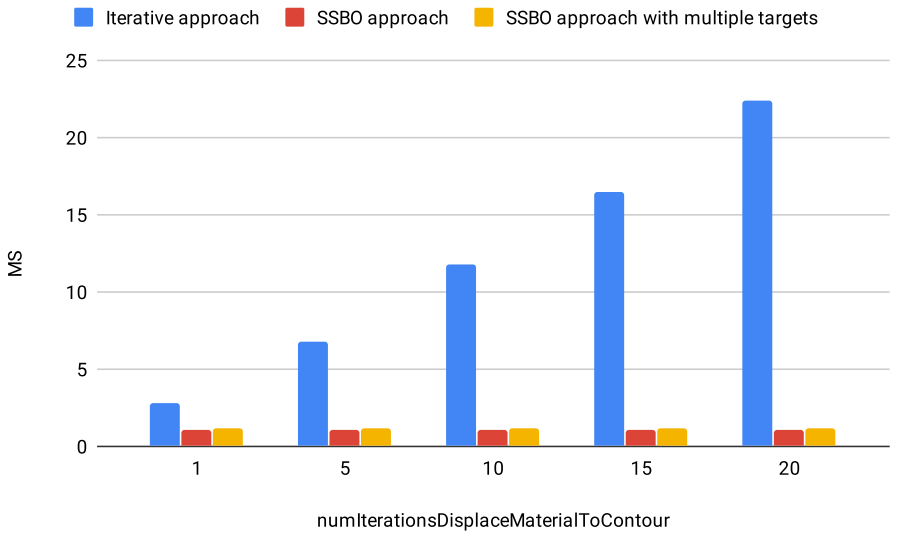


Figure 5.6: *Time consumption for displacing intersected material. The SSBO based approaches are not dependent on numIterationsDisplaceMaterialTo-Contour, but are shown in the plot for easy comparison.*

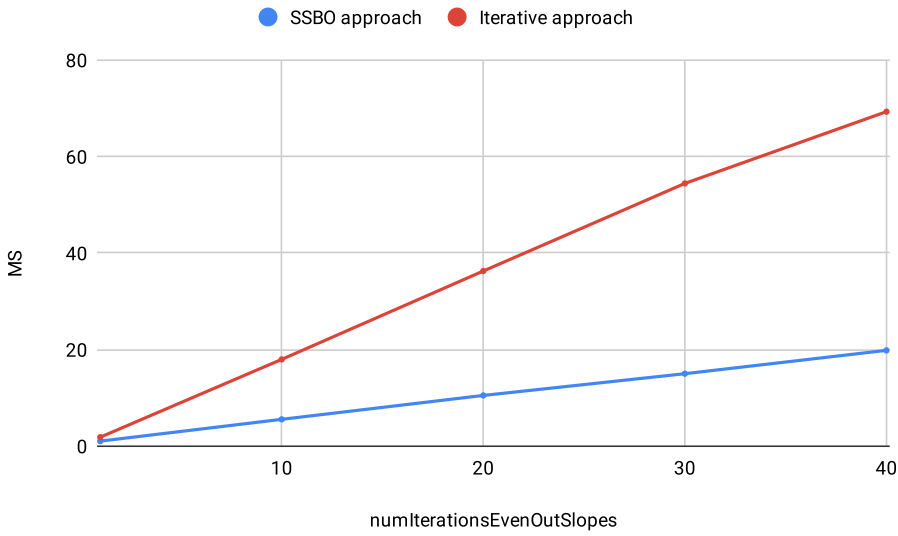


Figure 5.7: *Time consumption to even out steep slopes for the iterative and the SSBO based approach.*

5.4 Active areas

The use of active areas significantly increases the performance of the algorithm. The number of updated and simulated pixels is limited at the expense of an increased number of render calls. The number of render calls grows with the number of active areas used (which is based on the number of objects in the scene). This evaluation tries to show the benefit of using active areas, by varying the number of cars driving across the terrain. See Figure 5.8 for an illustration of the scene setup.

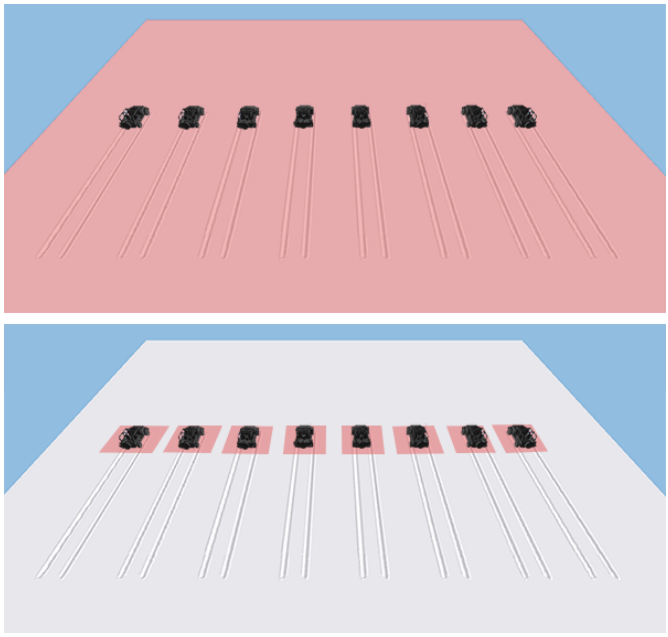


Figure 5.8: *Simulation of entire texture (top). Simulation of sub regions (active areas) (bottom). The highlighted areas illustrates which pixels are being updated each frame.*

Figure 5.9 shows the FPS with and without active areas enabled, using both the iterative and SSBO based method for the scene setup explained above. Table 5.7 shows the same measurements in tabular form. Table 5.8 shows the configuration used during the evaluation.

Simulating the entire texture corresponds to 4 194 304 pixels (2048 x 2048). In this evaluation, each active area covers 26 569 pixels. With eight cars in the scene, this corresponds to 212 552 pixels being simulated in total using the active areas.

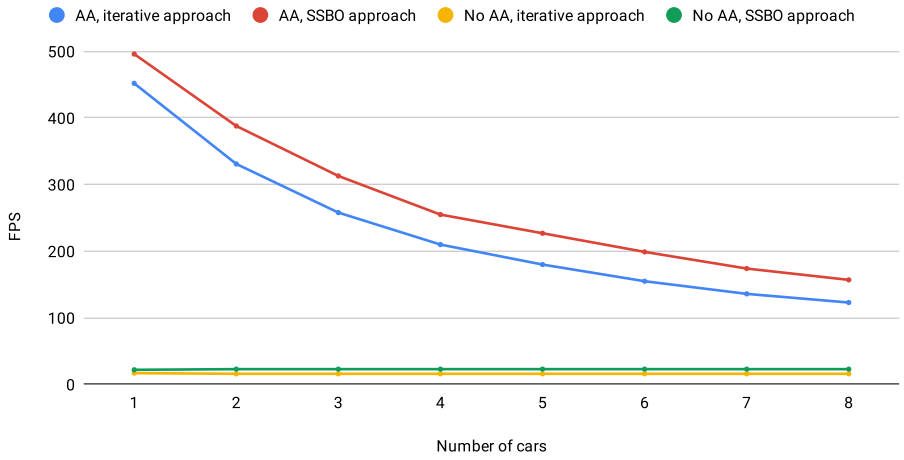


Figure 5.9: *FPS with and without active areas enabled for the iterative and the SSBO based approach. Active areas is abbreviated AA in the figure.*

Table 5.7: *FPS with and without active areas enabled for the iterative and the SSBO based approach.*

	Active areas enabled		**Active areas disabled**	
Number of cars	**Iterative**	**SSBO**	**Iterative**	**SSBO**
1	452	496	17	22
2	331	388	16	23
3	258	313	16	23
4	210	255	16	23
5	180	227	16	23
6	155	199	16	23
7	136	174	16	23
8	123	157	16	23

Table 5.8: *Configurations for active areas evaluation.*

Parameter	**Value**
textureSize	2048
terrainSize	300
numIterationsEvenOutSlopes	1
numIterationsDisplaceMaterialToContour	10

5.5 Volume preservation

To prove that the system preserves the terrain volume over time, a simulation with a car driving over the terrain is performed. At every 500th frame, the terrain heightmap texture is fetched from the GPU to the CPU. Next, each pixel is read and accumulated to a total summation of all pixels in the texture. *glReadPixels()* is used to transfer the texture from VRAM to RAM. Table 5.9 shows the result over the first 5000 frames. Figure 5.10 shows the scene setup. During the evaluation, active areas are turned off so that the entire terrain is simulated. Furthermore, compression is set to 0 since this is a prerequisite for volume preservation.

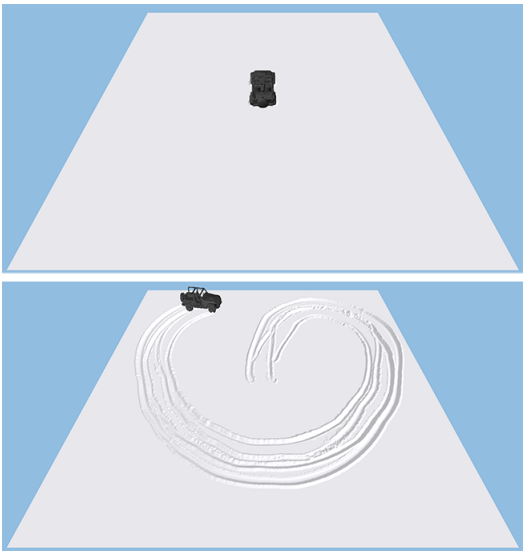


Figure 5.10: *Initial terrain (top). Terrain at end of simulation (bottom).*

Table 5.9: *Total terrain volume at different frames.*

Frame	Accumulated value
-	889 192 448 (initial value)
500	889 192 448
1000	889 192 448
1500	889 192 448
2000	889 192 448
2500	889 192 448
3000	889 192 448
3500	889 192 448
4000	889 192 448
4500	889 192 448
5000	889 192 448

5.6 Visual examples of deformations

This section provides some examples of deformations that can be achieved using different configurations. All examples use the SSBO based approach with multiple targets. The parameters that are varied are compression, roughness, slopeThreshold and heightmapSize.

Figure 5.11, 5.12, 5.13 and 5.14 shows different shapes of car trails in snow using different configurations. Figure 5.15 shows different shapes of foot trails. The different configurations produce different properties of the snow.

Figure 5.11: *HeightmapSize = 512 (left). HeightmapSize = 4096 (right). Common parameters: compression = 0, roughness = 0.6, slopeThreshold = 3.1.*

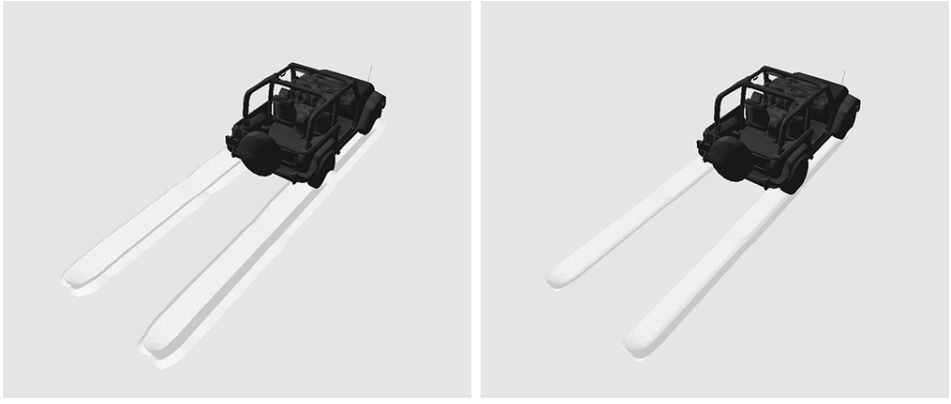


Figure 5.12: *Compression = 0 (left). Compression = 0.9 (right). Common parameters: roughness = 0.5, slopeThreshold = 1.9.*

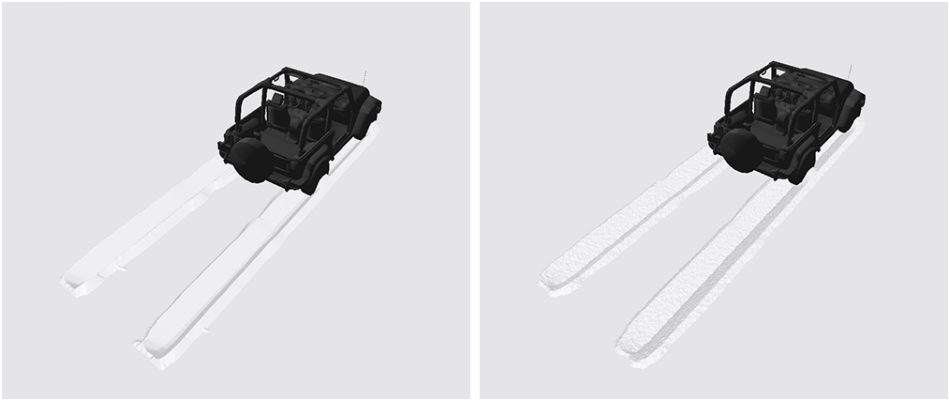


Figure 5.13: *Roughness = 0.1, slopeThreshold = 1.9 (left). Roughness = 0.8, slopeThreshold = 3.9 (right). Common parameters: compression = 0, slopeThreshold = 1.9.*

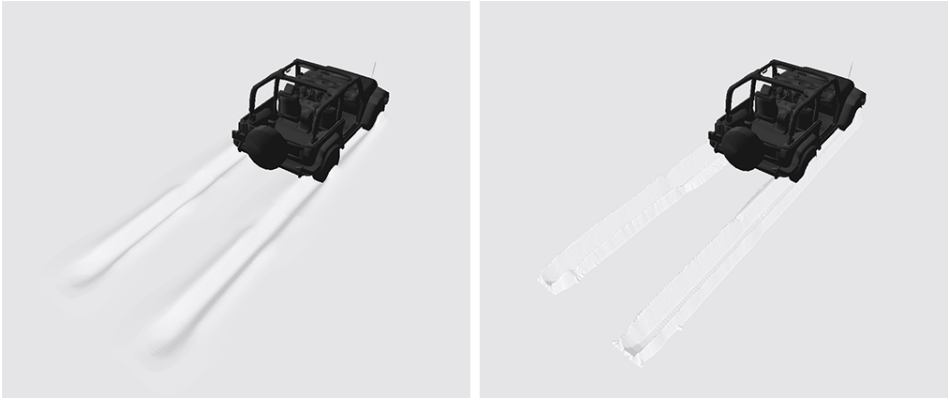


Figure 5.14: *SlopeThreshold = 0.1 (left). SlopeThreshold = 4.9 (right). Common parameters: compression = 0, roughness = 0.3.*

Figure 5.15: *SlopeThreshold = 8, compression = 0.95, roughness = 0.1 (left). SlopeThreshold = 8, compression = 0.95, roughness = 0.1 (middle). SlopeThreshold = 1.1, compression = 0.0, roughness = 0.1 (right).*

6 Discussion

This section discusses the result in terms of how it compares to other approaches, the performance of the different steps in the algorithm, limitations to the system and ideas for future work.

6.1 Performance

The result clearly shows that the SSBO based approach consistently outperforms the iterative approach in all measurements. The overhead of preforming arbitrary writes is significantly outweighed by reducing the number of render calls (both when displacing material and when evening out steep slopes).

When displacing material to the contour of objects, the iterative approach is significantly slower than the other methods, even when numIterationsDisplaceMaterialToContour is set to 1 (see Figure 5.6). The visual result differs among the different displacement methods and distance metrics. However, after evening out the slopes the difference is less noticeable.

Figure 5.7 shows that the SSBO based approach to evening out steep slopes is more than twice as fast as the iterative approach. This could be attributed to the use of two shader programs that are interleaved for the iterative approach. This requires twice as many render calls and some overhead from binding a new shader programs twice per iteration.

When looking at the time measurements for all steps of the SSBO based approach in Figure 5.5, it is clear that the Jump Flooding shader program incurs the highest performance impact. The reason for this is presumably a combination of the fact that it requires 12 render calls on a heightmap size of 4096 and that it reads from

a texture with a pixel size of 128 bits. The algorithm could be updated to only use a two-channel texture with 16 bits per channel for the Jump Flooding algorithm, and then perform an additional render pass to combine the Jump Flooding result with the penetration texture. However, as seen in Figure 5.4 and 5.5, it is when the heightmap size reaches 4096 that the performance hit is incurred.

The system is shown to be volume-preserving. However, this is achieved using integer textures, which comes with some drawback. The first is a lack of linear sampling, which is solved by introducing a render call that moves the integer data to a float texture. Another is the need for 32 bits when storing height values. Using 16 bit, an unsigned integer can store values between 0 and 65 535. However, to achieve floating point precision when deforming and rendering, this value is divided by 10 000. This would allow a very limited height range of 0 to 6.55. On the other hand, if volume preservation is not a requirement, floating point textures with 16-bit channels for storing heights could be used. Therefore, using floating point textures would probably improve the performance significantly, as memory bandwidth is a bottleneck on today's GPUs.

The SSBO approach with multiple targets ensures that intersected material is moved to multiple columns, instead of just the columns around the object. This ensures that the contour doesn't become as large, which means that the number of iterations to even out steep slopes can be reduced. As seen in Figure 5.6, the performance of distributing to multiple columns is only slightly worse than when all material is moved to one column. However, as seen in Figure 5.7, the performance impact of evening out steep slopes quickly increases as numIterationsEvenOutSlopes is increased. Therefore, the additional performance hit of distributing to multiple columns might be worth it if numIterationsEvenOutSlopes can be reduced.

6.2 Future work

In this section multiple areas for future work are discussed.

The use of obstacle maps in the algorithm makes sure than an object cannot push material through itself or other objects when intersecting material. Generally, this ensures that material is displaced in the direction the object travels, which addresses a shortcoming of [16]. However, if objects are very thin or travel at very high speeds, such that between two frames, there is no overlap in the obstacle map, the deformation algorithm would interpret that as a teleportation rather than a movement. This would allow material to be displaced uniformly around the object, as opposed to in the direction of travel. To solve this, it might be possible to interpolate between the current and the previous obstacle map. Another approach would be storing the velocities of objects in the depth texture when it is created. The velocity could then directly be used when displacing material.

Another limitation to the obstacle map is how it behaves for objects that are above the terrain. The decision whether an object is an obstacle or not is binary. How-

ever, an object might be well above the terrain, but if it is surrounded by material at a higher altitude, this material should not be able to be displaced through the object. A suggestion is to use the difference between the obstacles' height and the terrain's height to calculate how much material each column can receive. However, this would make the displacement algorithm more complex. Each column might need to keep track of multiple targets, in case all of its material does not fit in one column.

A third limitation to the current implementation is that there is no collision detection between the terrain and objects that limits how deep an object can penetrate the ground. Some type of collision detection is necessary to realistically drive a car across a bumpy terrain, for instance. In [4], the modified terrain was streamed back to the CPU to be used in collision detection. This would not necessarily have to be performed each frame. Furthermore, the texture could be rendered to a lower resolution texture before reading it back to the CPU. That way, the collision detection and transferring the pixels to the CPU would be faster, at the expense of less detailed collision detection.

While the deformations are persistent on the terrain, there is no memory of how much the material has been compressed from previous deformations. If a car drives over a patch of terrain, and the compression is set to 0.5, half of the material would be displaced, while the other half would be removed to simulate it being pressed down (compressed). However, by storing the compression of each column in a texture, it would be possible to model material more realistically. One example is that the compression could be used in the collision detection, so that if a column is highly compressed, the collision detection takes that into account and limits how much the object can penetrate the ground.

The use of active areas significantly increases the performance, as shown in Figure 5.9. However, there is a big performance impact of increasing the number of render calls. Future work could focus on searching for nearby active areas and combine them to bigger areas, to limit the number of render calls. The decision to combine active areas would probably depend on how distant they are and their sizes.

Another limitation is that only one heightmap texture is supported, this limits the ability to render huge terrains. Figure 5.9 showed that almost 500 FPS was reachable on a 2048 x 2048 heightmap with one active area. However, the terrain was not very big. While the terrain size can be increased while keeping the heightmap size constant, this leads to less detailed deformations, as show in Figure 5.11. To add support for huge terrains, a chunk-based LOD-scheme could be used, similar to that of [17]. This would divide the terrain into multiple meshes, at different resolutions. Furthermore, the work by [19] could be used to allow deformations on terrains that are divided into multiple heightmap textures at different resolutions. While this would increase the memory usage, it would allow the deformation system to work on huge terrain with highly detailed deformations, potentially at similar performance as shown in this report.

7

Conclusion

In this report, a system for volume-preserving, persistent and dynamic terrain deformation in real-time has been presented. Volume-preserving refers to the volume of the terrain heightmap being consistent over time as the terrain is deformed. Persistent refers to the fact that deformations are kept in memory for the lifetime of the application. Dynamic refers to the support of deformations from any shape of colliding objects as well as the ability to produce different deformation shapes in the terrain through multiple user parameters.

The deformation system has been implemented using different algorithms. One is limited to static write locations in the shader, while the other two approaches take advantage of recent features such as SSBOs to allow arbitrary write locations. The system and the different approaches have been evaluated in terms of the performance impact and the visual outcome.

The end result is a real-time performing terrain deformation system that takes advantage of the parallelism of GPUs to efficiently find objects that intersect the terrain. Material that has been intersected is displaced and animated, based on the direction and speed of which the objects intersect the terrain. The system achieves high enough performance to be used in real-time applications such as games and simulators that strive for realistic interaction between the terrain and objects. Furthermore, multiple aspects for future work are proposed. This includes adding a collision detection system and supporting deformation on huge terrains by utilizing a level-of-detail scheme for the terrain heightmap.

Bibliography

[1] Johan Andersson. Terrain rendering in frostbite using procedural shader splatting. In *ACM SIGGRAPH 2007 courses*, pages 38–58. ACM, 2007.

[2] Anthony Aquilio, Jeremy Brooks, Ying Zhu, and Owen Scott. Real-time gpu-based simulation of dynamic terrain. *Advances in Visual Computing : Second International Symposium, ISVC 2006 Lake Tahoe, NV, USA, November 6-8, 2006 Proceedings, Part I*, page 891, 2016.

[3] Colin Barré-Brisebois. Deformable snow rendering in batman: Arkham origins. In *Game Developers Conference (GDC), San Francisco, California, March*, 2014.

[4] Justin Crause, Andrew Flower, and Patrick Marais. A system for real-time deformable terrain. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment*, pages 77–86. ACM, 2011.

[5] Alexander Frisk. Real-time vehicle trails in deformable terrain. Master's thesis, Umeå University, 2017.

[6] Toby Heyn. Simulation of tracked vehicles on granular terrain leveraging gpu computing. Master's thesis, University of Wisconsin–Madison, 2009.

[7] Daniel Holz, Thomas Beer, and Torsten Kuhlen. Soil deformation models for real-time simulation: a hybrid approach. In *Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS*, 2009.

[8] Anton Kai Michels and Peter Sikachev. Deferred snow deformation in rise of the tomb raider. In Wolfgang Engel, editor, *GPU Pro 360 Guide to Geometry Manipulation*, chapter 18. CRC Press, 2018.

[9] Koichi Onoue and Tomoyuki Nishita. An interactive deformation system for granular material. In *Computer Graphics Forum*, volume 24, pages 51–60. Wiley Online Library, 2005.

[10] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[11] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116. ACM, 2006.

[12] Witawat Rungjiratananon, Zoltan Szego, Yoshihiro Kanamori, and Tomoyuki Nishita. Real-time animation of sand-water interaction. *Computer Graphics Forum*, 27(7):1887 – 1893, 2008.

[13] Henry Schäfer, Benjamin Keinert, Matthias Nießner, Christoph Buchenau, Michael Guthe, and Marc Stamminger. Real-time deformation of subdivision surfaces from object collisions. In Wolfgang Engel, editor, *GPU Pro 6: Advanced Rendering Techniques*, chapter 3. CRC Press, 2016.

[14] Dave Shreiner, Graham Sellers, John Kessenich, and Bill Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

[15] Jean-François St-Amour. Rendering assassin's creed iii. Game Developer Conference, 2013. URL `https://www.gdcvault.com/play/1017710/Rendering-Assassin-s-Creed`.

[16] Robert Sumner, James O'Brien, and Jessica Hodgins. Animating sand, mud, and snow. In *Computer Graphics Forum*, volume 18, pages 17–26. Wiley Online Library, 1999.

[17] Thatcher Ulrich. Rendering massive terrains using chunked level of detail control. In *Proc. ACM SIGGRAPH 2002*, 2002.

[18] Dong Wang, Yunan Zhang, Peng Tian, and Nanming Yan. Real-time gpu-based visualization of tile tracks in dynamic terrain. pages 1–4, 2009.

[19] Egor Yusov. Real-time deformable terrain rendering with directx 11. In Wolfgang Engel, editor, *GPU PRO 3: Advanced Rendering Techniques*, chapter 2. CRC Press, 2012.