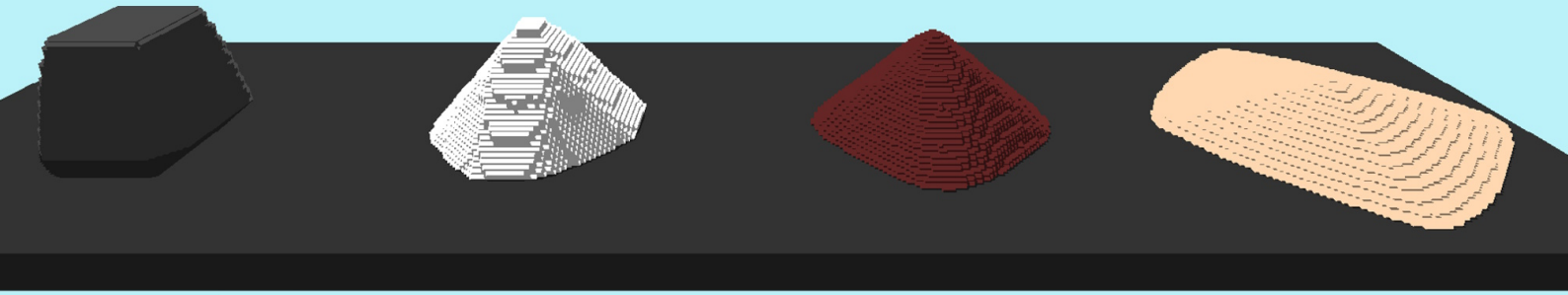
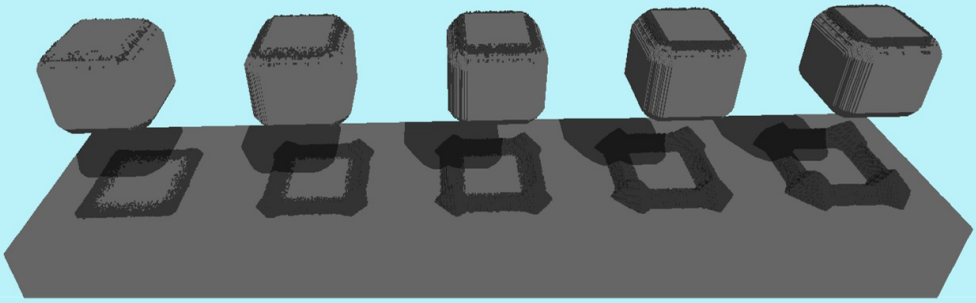
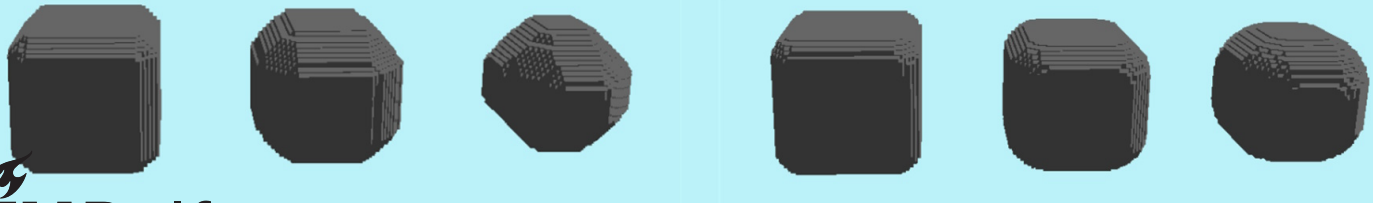
Weathering and Debris Simulation on High-Resolution Voxel Scenes

M. Kuijpers







TU Delft

Weathering and Debris Simulation on High-Resolution Voxel Scenes

by

M. Kuijpers

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday June 14, 2023 at 13:00.

Student number:	4575075	
Thesis committee:	Prof. Dr. E. Eisemann,	TU Delft, supervisor
	Y. Dijkstra,	TU Delft
	M. Billeter,	University of Leeds

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Acknowledgements

I would like to express my gratitude to the Delft University of Technology for the completion of this thesis. Especially thanks to my supervisor Prof. Dr. Elmar Eisemann, and Mathijs Molenaar for the continuous feedback over the last period. Also thanks to my thesis committee, Markus Billeter and Yoeri Dijkstra.

And finally, many thanks to my friends and family for their continuous support throughout the whole period. Without your support, it would have been a much harder feat than it already turned out to be. Thank you.

Abstract

Voxels are cells on a 3D regular grid. Voxel-based scenes have many applications, including frequent use in simulations or games. Over the years, the field of high-resolution voxel scenes has progressed significantly, allowing for compressing and real-time editing of high-resolution scenes. This possibility of editing high-resolution scenes led to various editing tools. This thesis aims to expand this set of tools with a focus on more complex simulation-based solutions. We explored the field of terrain weathering and decided on a spheroidal weathering tool to perform weathering on voxel-based scenes. One of the existing limitations of this particular method was that it did not account for debris simulation. We ultimately decided to add a granular simulation using a layer-based approach that integrates with the weathering tool but is also able to function in a standalone manner.

This thesis presents three highly-customizable editing tools for voxel-based scenes: spheroidal weathering, granular simulation, and a combined tool that enables weathering with debris simulation. These tools are integrated into the HashDAG framework, which is a high-resolution voxel-grid method building upon a directed acyclic graph representation of a sparse voxel octree. Our solutions enable close to real-time editing for changes with a smaller regional support. There is room for improvement in terms of performance at scale, with various potential ideas presented in the future work section.

Contents

List of Figures

1 Introduction

In the field of computer graphics, voxel-based scenes are common. These scenes are constructed by voxels, which are cells on a 3D regular grid. Storing whether a voxel is present for each position in a grid is memory inefficient, as large regions of the scene are often empty - in other words, the scene is sparse. Compression techniques have enabled high-resolution voxel scenes by reducing memory usage for large empty regions [16] and repetition [8, 14], which is necessarily common in binary high-resolution grids. The HashDAG framework Careil et al. [5] builds upon the latter techniques and is a framework to perform real-time edits of compressed voxel scenes with various types of editing operations such as adding or removing primitive shapes and coloring voxels. Terrain weathering tools are often used to create aged, worn-out, or weathered objects, adding more realism to the terrain and allowing more artistic freedom for artists. This thesis aims to expand the set of editing tools using terrain weathering, allowing for erosion and decay of terrain over time.

Many works aim to simulate terrain weathering. Our focus is on on voxel-based weathering techniques as we want to extend voxel-based editing tools. Many voxel-based weathering methods exist [1, 9, 10, 13]. One of these is spheroidal weathering [1], a method that removes sharp edges and corners of a scene quicker than rounder surfaces, smoothing out terrain. We decide to utilize this weathering method for a customizable weathering tool because it can be applied on voxels, and because of its customization options. Additionally, we implement this on the GPU using [1] as this method is well-suited for parallelization.

When terrain weathers in real-life, removed material falls down or gets by transferred by air or water. Simulating this material aids the realism of weathering methods to more accurately represent the real-life behavior of weathered material. It can also help understand the effects of weathering on the surroundings when simulating debris material. Therefore, we also explore the field of debris simulation to simulate the weathered material of the weathering tool.

Some methods allowed for debris simulation in combination with spheroidal weathering, however, we ultimately decided not to use these because of various reasons. Therefore, we decided to focus on other simulation methods. The term granular simulation is often used as a more general term to describe simulating smaller pieces of material. We ultimately decide on the Arches simulation method [21]. Because this method uses a layer-based simulation method, the method can be utilized in a voxel-based editing tool by converting voxels into granular material layers. It also allows for simulating material under overhangs and allows for customizable material types with different repose angles. Because of these reasons, we decide to use this method to create a customizable granular simulation tool.

[1] NVIDIA CUDA Toolkit: `https://developer.nvidia.com/cuda-toolkit`

Because this granular simulation method works by converting voxels into material layers, we decide to use it to integrate with the weathering tool. All weathered voxels are converted into granular material layers to simulate debris material. This combined editing tool can be used to weather material and simulate the debris afterward. Because a lot of debris material can be created, we also allow for customization of the amount of debris material that is created.

This thesis extends the existing editing tools of the HashDAG framework, with the main contributions being:

1. Adaption of the spheroidal weathering method to integrate the Spheroidal Weathering tool in the HashDAG framework, implemented on the GPU using CUDA[2].

2. Adaption of the spheroidal weathering method to integrate of the Arches Granular simulation tool in the HashDAG framework.

3. Integration of a novel editing tool by combining the Arches granular simulation method and Spheroidal Weathering method to perform debris simulation for the Spheroidal Weathering method. Solely the Spheroidal Weathering part is implemented on the GPU.

Chapter 2 discusses related work of voxel scenes, terrain weathering, and granular simulation. Chapter 3 discusses in-depth background information on the HashDAG framework, Spheroidal Weathering, and the Arches simulation method. Chapter 4 discusses the details of the new proposed editing tools. Chapter 5 presents information about the implementation details for these tools. Chapter 6 presents the results of the editing tools, including examples, benchmarks, and visualizations of customization parameters. Chapter 7 presents some discussion for some topics. Chapter 8 discusses the limitations. Chapter 9 presents the conclusions, and finally, Chapter 10 discusses ideas and suggestions for future work.

[2]NVIDIA CUDA Toolkit: `https://developer.nvidia.com/cuda-toolkit`

2

Related Work

Three different fields of research are relevant for this thesis; voxel scenes, terrain weathering, and granular simulation. In this section, we will discuss relevant published work for each of these fields. For the field of voxel scenes we already established that we will be using the HashDAG framework, so we will present the papers leading up to this work.

2.1. Voxel Scenes

In recent years, quite some progress has been made regarding high-resolution voxel scenes. First, Meagher [16], introduce a sparse octree encoding to represent voxel data. This encoding exploits sparsity in a regular 3D by recursively subdividing a grid and only storing parts containing data (Figure 2.1). Kämpe et al. [14] further improve voxel scene compression with a sparse voxel Direct Acyclic Graph (DAG) encoding by exploiting similarity in a scene. This method collapses an octree structure by merging equivalent subtrees (Figure 2.2). Dado et al. [8] add attribute compression, allowing for the compression of arbitrary data (e.g. color) in sparse voxel DAGs (Figure 2.3). Careil et al. [5] present a method to interactively edit compressed sparse voxel DAGs in real-time, compressing geometry but not colors (Figure 2.4).

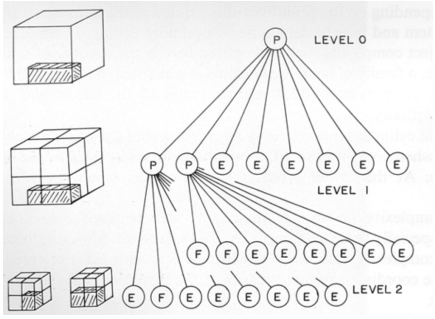


Figure 2.1: Sparse octree encoding by Meagher [16]

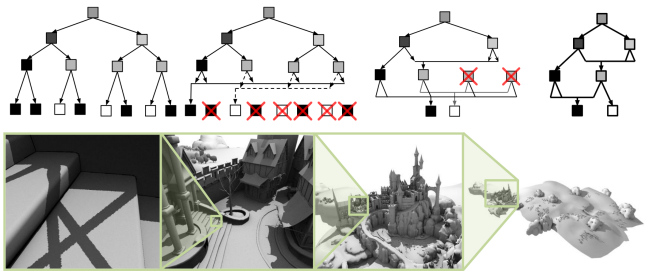


Figure 2.2: Sparse voxel DAGs reusing equal subtrees, by Kämpe et al. [14]

Figure 2.3: Dado et al. [8]

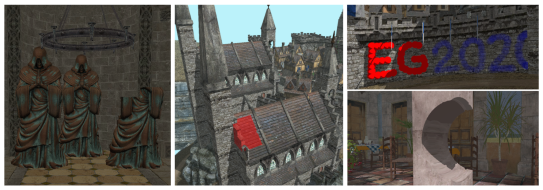


Figure 2.4: Careil et al. [5]

2.2. Terrain Weathering

Terrain weathering refers to the process of simulating the natural decay of terrain over time through the effects of, e.g., wind or water. Chen et al. [6] present a visual terrain weathering method with an additional step to deform geometry (Figure 2.5). Their method traces so-called γ-ton rays that reflect, bounce, flow, and settle on the terrain. These settled γ-tons are used to displace the geometry. It has visually pleasing results with visual weathering (by changing the texture colors) and eroded terrains. However, the method is relatively slow with iterations taking minutes to execute, and tens of iterations are often necessary to achieve the desired result.

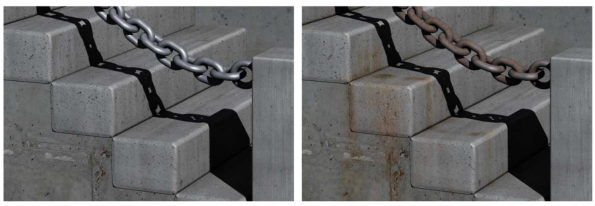


Figure 2.5: Chen et al. [6]

The work of Beardall et al. [1] transforms a 3D mesh into a voxel representation and performs weathering on the voxel representation (Figure 2.6). The voxel representation is transformed back into a 3D mesh after weathering. This method is much more applicable for voxel-based scenes because it performs weathering on a voxel representation. The method is iterative and attempts to simulate spheroidal weathering phenomena based on terrain curvature. Results are often visually pleasing and physically feasible. Jones et al. [10] directly extended this work with colluvium deposition and cavernous weathering. Colluvium deposition is the term used for debris simulation of weathered terrain, and cavernous weathering enables weathering for terrain with negative curvatures, such as caverns (Figure 2.7). The work by Farley [9] further extends on both the spheroidal [1] and cavernous weathering [10] with the addition of a GPU implementation for the weathering and colluvium deposition simulation (Figure 2.8). Both methods simulate individual falling voxels. Farley [9] implemented this on the GPU, however, the runtime of this method is still slow, within the range of tens of seconds per step while requiring multiple tens of steps to complete. This method also simulated each individual falling voxel, which requires checking all voxels below each falling voxel, which can become costly depending on the data structure. If a buffer is used, this can be quick, however for example with a tree data structure, this can become more costly.

Figure 2.6: Beardall et al. [1]

Figure 2.7: Jones et al. [10]

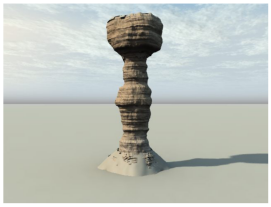


Figure 2.8: Farley [9]

Work of Krs et al. [13] presents a weathering method that uses a Smoothed-Particle Hydrodynamics (SPH) simulation [18] to simulate wind as a weathering force (Figure 2.9). A mesh is transformed into voxels with a combination of material stacks to define material properties. Erosion is simulated by converting surface voxels into material particles and simulating those particles using fluid dynamics, where these particles further convert voxels into material particles. Stabilized material particles are converted back into voxels to simulate material deposition. However, due to the increased complexity of using a particle simulation (the SPH), we decide to not use this method.

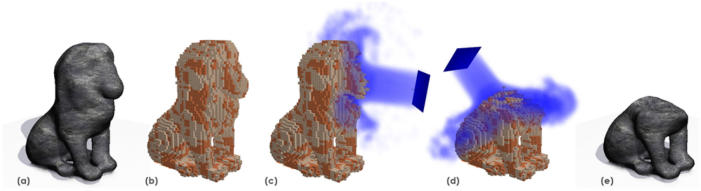


Figure 2.9: Krs et al. [13]

We chose to use the work of Beardall et al. [1] as a base for the weathering tool. The presented method transfers well to a voxel-based editing tool, with the potential to expand it with debris simulation.

The other works that present debris simulation methods do present useful solutions[9, 10], albeit these debris simulations are not as performant. They are also not as widely applicable because they are specifically tailored to the method proposed by Beardall et al. [1]. Because the focus of this thesis is to expand the HashDAG framework with editing tools, a granular or debris simulation method that can be used in isolation as a standalone tool and for debris simulation is preferred over other narrow tailored solutions.

2.3. Erosion/Granular Simulation

Only some of the previously mentioned terrain weathering works simulate debris material after weathering. We explored various methods of simulating granular material that could be relevant for voxel-based scenes. The works we discuss here aim to simulate or utilize the geological concept of erosion, the process of gradual removal or displacement of material due to various causes by nature. In particular, hydraulic and thermal erosion is used in the works we cover. Hydraulic erosion is caused by the transfer of material with water from rivers or rain. Thermal erosion is caused by changes in temperature, leading to cracks that decompose into smaller parts.

2.3.1. Heightmap-based

Many early works make use of heightmaps to represent scenes and to simulate material and erosion. Heightmaps are 2D grids with a value to indicate the height at that position. For example, the work of Neidhold et al. [19] simulates hydraulic erosion using a heightmap for terrain and water height, including some other 3D information such as acceleration and velocity (Figure 2.10). The work of Zeng et al. [23] uses heightmaps and allows moving objects in the scene to deform this terrain. (Figure 2.11). Krištof et al. [12] use a completely different approach by using Smoothed-Particle Hydrodynamics (SPH) [18] to simulate hydraulic erosion and deposition, using particles to deform heightmaps (Figure 2.12). O'Brien and Hodgins [20] introduce a simulation method based on virtual pipes, which is parallelized by Mei et al. [17] to simulate hydraulic erosion using multiple heightmaps for terrain, water, and sediment (Figure 2.13). To follow up this work of Mei et al. [17], Jákó [11] further extends it using a thermal erosion model of Beneš and Forsbach [2]. This results in a more realistic simulation result (Figure 2.14).

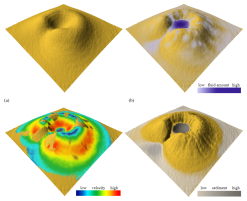


Figure 2.10: Neidhold et al. [19]

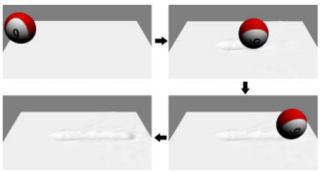


Figure 2.11: Zeng et al. [23]

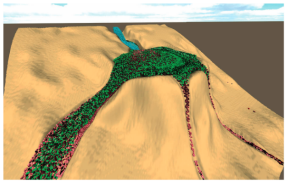


Figure 2.12: Krištof et al. [12]

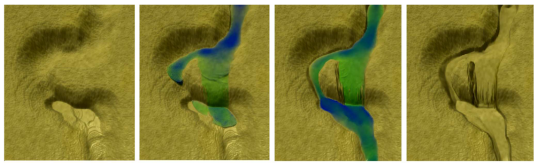


Figure 2.13: Mei et al. [17]

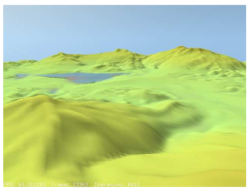


Figure 2.14: Jákó [11]

2.3.2. Layered representation

The work of Beneš and Forsbach [2] introduced a layered data representation (Figure 2.15). Here, a 2D grid is used to store stacks of material as opposed to just the height with heightmaps. This allows a single stack to have different materials in a single position in the grid. This layered data representation is used to simulate thermal erosion, where material stabilizes under some angle called the repose angle. This early work was able to fairly accurately simulate thermal erosion using a simple data structure and serves to be a basis for later works.

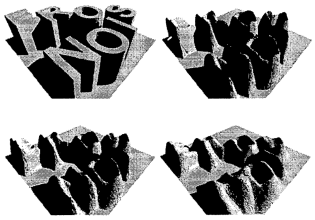


Figure 2.15: Beneš and Forsbach [2]

Follow-up work by Beneš and Forsbach [3] used this layered data structure to simulate hydraulic erosion (Figure 2.16). It utilizes water layers capable of absorbing sediment material. By simulating the movement of these layers, sediment is moved. Finally, water layers slowly evaporate and deposit the absorbed sediment material. Beneš and Roa [4] also proposed another method utilizing the stacked data structure; this time to simulate sand behavior in desert scenery (Figure 2.17). It presents methods to generate wind ripples and shadows, as well as the accumulation of sand on obstacles. Peytavie et al. [21] further extended the layered data structure simulation via different material types, like solid immovable layers and air layers, forming overhangs in scenes (Figure 2.18). Additionally, this work introduced different granular material types with other repose angles. To simulate these new material types, the simulation was slightly adjusted. All layers in between solid layers are sorted and then merged together, such that there are no interweaving granular layers of different types.

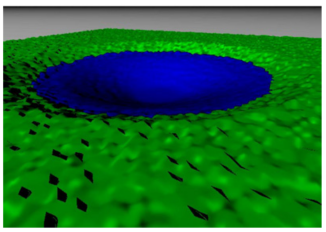


Figure 2.16: Beneš and Forsbach [3]

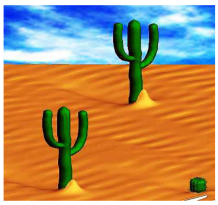


Figure 2.17: Beneš and Roa [4]

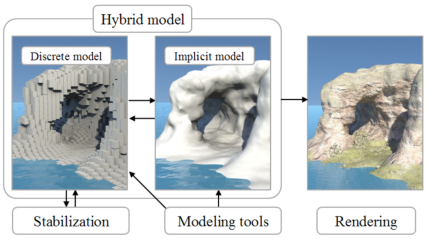


Figure 2.18: Peytavie et al. [21]

Chentanez and Müller [7] also uses a layered data structure, however now to simulate water on the GPU (Figure 2.19). It used the layers somewhat differently than the other layered data structure methods, however. By design, it makes use of three 'layers'. A single stack in this method always consists of a single base terrain layer, a single tall layer, and finally a fixed number of regular cells with a height of one. This tall cell is used to model the part of the water under the surface, not requiring detailed simulation, only a height. This method simulates stacks and cells with a hierarchical structure in real time on a GPU. It is not capable of simulating multiple material types, however, because by design, the bottom layers are always a terrain and a tall layer and subsequent layers are all regular cells stacked upon each other. It does not allow for any other layers other than those three.

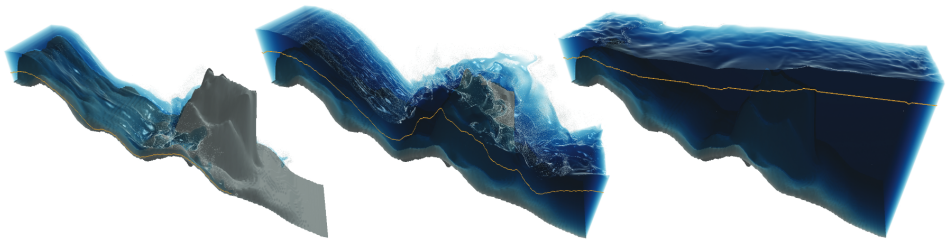


Figure 2.19: Chentanez and Müller [7]

Considering the previous works on granular simulation, we decided to use the method by Peytavie et al. [21] and create a debris simulation, which we integrate with the method of Beardall et al. [1]. Additionally, we use it to create a separate granular simulation tool that can be used outside of the weathering tool. The choice also stems from the fact that heightmap methods are not applicable to voxel-based scenes because they do not allow for simulations containing overhangs. Among the layer-based simulation methods, the method from Peytavie et al. [21] is very flexible, supports overhangs, and has various customization options for granular material types.

3

Background

As discussed in Chapter 2, we use the HashDAG framework of Careil et al. [5] as a base for creating new voxel editing tools. We use the spheroidal weathering method of Beardall et al. [1] to create the weathering tool. We use the Arches simulation method of Peytavie et al. [21] to create the debris simulation for the weathering tool, on top of a standalone granular simulation tool. This section discusses in-depth background information for these works and their relation to the proposed method.

We discuss the three main background methods. Section 3.1 discusses the HashDAG framework of Careil et al. [5]. Section 3.2 discusses the Spheroidal Weathering method based on Beardall et al. [1]. Section 3.3 discusses the Arches simulation method based on Peytavie et al. [21]. Finally, Section 3.4 covers details of NVIDIA CUDA, relevant to our implementation of the Spheroidal Weathering tool.

3.1. HashDAG

This section summarizes the work of Careil et al. [5]. We cover the HashDAG framework and discuss why the HashDAG framework is a good suitable point for creating the voxel-based editing tools introduced in the thesis.

3.1.1. Contributions

The HashDAG framework extends different works of high-resolution voxel scenes and enables editing of high-resolution compressed voxel scenes [8, 14–16]. Its three main contributions are:

1. A method for modifying sparse voxel DAGs in their compressed form, without decompression and recompression.

2. A novel data structure that enables interactivity and has little impact on compression, rendering, and traversal performance.

3. A recording of changes that enable undo/redo operations with associated garbage collection to free up memory when needed.

3.1.2. Method

This subsection summarizes the proposed methods of Careil et al. [5]. We cover how DAG geometry is modified, the HashDAG structure, and its editing tools.

Modifying DAG Geometry An assumption is made that edits take place within a bounding volume of a cube in 3D space. First, voxel locations that require change are identified. This is done by traversing the DAG in a depth-first manner. If a node is not affected (i.e., it is outside the bounding volume), the traversal is terminated. If a node in the volume does not exist, the DAG structure is edited and new nodes are added. Finally, if an entire region of a subtree is encapsulated by the edit bounding volume, an early termination might be possible.

During the depth-first traversal, whenever the depth and location of manipulation for the edit operation are reached, a new node is created to represent the modified geometry. It is integrated into the DAG either by reusing an identical leaf-subvolume if it exists or by appending the new node to the DAG. Next, the changes are propagated up to the parent node. A special case is where the node ends up empty, then the parent omits that node from its child mask to indicate its absence. Once all children of a parent node have been processed, a new node is created to reflect the changes with a set of pointers to all non-empty children and a bit-mask of non-empty children. If this particular node already exists in the DAG, the node is reused, otherwise, the new node is added to the DAG. Both cases result in a pointer to a new node that is passed up to the parent.

Edit operations are repeated up to the root node. To represent the new geometric changes, a new root node is created. The previous root node is kept, and trees attached to this previous root node reflect previous versions of the DAG, enabling undo and redo operations by exchanging the root node.

HashDAG structure The HashDAG structure makes use of two components: (1) hashing to efficiently find nodes, and (2) virtual memory to manage overall memory usage.

Each node can contain eight children with a children mask to define which of the children are set, each of these children are pointers to other nodes. A hash is calculated based on the children and the children mask, for in-depth details on the hashing method, refer to [5].

The HashDAG data structure is designed such that the data of all nodes in the HashDAG structure does not need to be moved by allocating a fixed address space for the whole DAG. Each level in the DAG has a predetermined portion of the address space, where less of the address space is allocated close to the root because these levels contain fewer nodes. To allow for faster searching of nodes, the address space of each level is further subdivided into buckets, where the hash of a node determines into what bucket a node is placed. By performing a linear search over all nodes in the corresponding bucket, nodes can be found quickly. A key property of the HashDAG is that hashing is only performed during editing. During traversal, no hashes need to be computed.

3.1.3. Editing tools

The work by Careil et al. [5] provides the following collection of editing tools to demonstrate the HashDAG and attribute modification:

- **Cube/Sphere**: Adds a solid voxel cube/sphere to the scene.
- **Sphere noise**: Adds voxels in the shape of a sphere with noise applied.
- **Sphere paint**: Paints all voxels in a spherical region to some color.
- **Cube copy**: Copies voxel terrain from a selected region to some other destination region.
- **Cube fill**: Flood fill and color voxels from the center of a cube region outward in the region of a cube.

3.2. Spheroidal Weathering

This subsection summarizes the most relevant contributions of the work of Beardall et al. [1]. It presents a novel method for weathering material in order to recreate goblin-like terrain structures, which are the structures shown in Figure 3.1.

3.2.1. Contributions

The main contribution of Beardall et al. is an efficient algorithm for generating concave terrain in a voxel scene. It uses the mathematical model from Sarracino et al. [22] for spheroidal weathering, a geological process where material with sharp edges and corners is weathered and reduced to a more rounded surface. The method focuses on generating sandstone goblin structures with visually plausible results and minimal artistic and technical effort. An example is shown in Figure 3.1.

(a) (b)

Figure 3.1: Goblin created using spheroidal weathering (a), and reference photograph (b). [1]

3.2.2. Weathering Model

This subsection summarizes relevant components of the weathering model as proposed by the original paper [1].

Weathering rate The dominant force by which terrain is weathered and goblins are formed is the geological process of spheroidal weathering. This process causes sharp edges of rock to erode quicker than flat parts. Here, we refer to the concept of weathering as the removal of material from a given structure or mesh. The shape of the structure causes parts of it to weather at different rates, resulting in non-uniform erosion over time. The rate at which this occurs is called the weathering rate. This rate is modeled using the surface area and volume of the rock in local areas. This so-called weathering rate r is proportional to the rock surface area a divided by the rock volume v in a local region around that location:

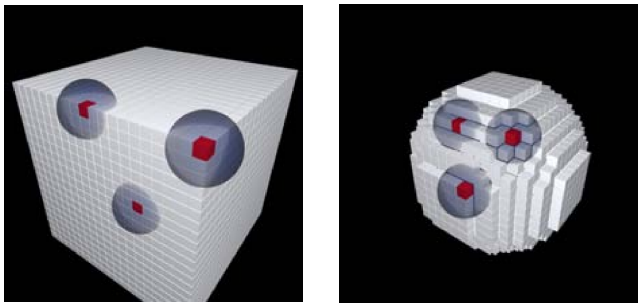
$$r \propto \frac{a}{v}$$

The original method of [1] made use of a mesh input, so first this mesh is voxelized. When using a mesh, the concept of weathering refers to the removal of voxels from the scene. The original method deems that when using a scene of discrete units, the weathering rate r is proportional to the number of exposed faces in a local area. Subsequently, when using voxels, this is proportional to the ratio between the number of air (A) and rock (R) voxels in a local region.

$$r \propto \text{number of exposed faces} \propto \frac{A}{R}$$

When referring to 'weathering' in relation to individual voxels, this indicates that voxels are removed as a result of a high air-to-rock ratio.

Bubbles To model the previously mentioned local regions, a structural volume, such as a sphere, cube, or other is introduced, typically referred to as 'bubbles'. Centered at a given position, their interior defines where the air-to-rock ratio and, thus, the weathering rate is computed. Figure 3.2 visualizes bubbles with spherical volumes around some voxels. The bubble on the corner in Figure 3.2 has a much higher air-to-rock ratio than voxels on the side, causing them to weather faster. The result of weathering can be seen in Figure 3.2b, where the sides of the cube have not moved much, but the corners of the cube have weathered more drastically.

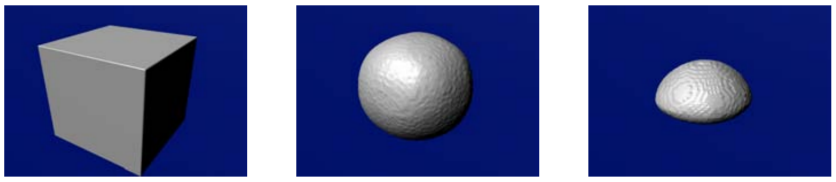


(a) (b)

Figure 3.2: (a) Before and (b) after image of weathering process with spherical bubble visualizations.[1]

Impact of bubble shape Any 3D volume can be used for bubbles, to get different weathering behavior. For example, to weather terrain more vertically, a taller spherical shape can be used. This causes the air-to-rock ratio and therefore the decimation rate of parts on the top and bottom of the terrain to be higher, and therefore weather more quickly.

Figure 3.3 visualizes the impact of different bubble shapes. Figure 3.3a shows the original structure, before any weathering. Figure 3.3b shows the original structure after weathering is applied for some number of iterations using a spherical bubble, resulting in a spherical shape. Figure 3.3c shows the structure after the same weathering procedure is performed with a teardrop bubble shape - a shape similar to a sphere but with additional volume on top. This teardrop shape has more volume on top, so when calculating the air-to-rock ratio, more air voxels from above are taken into account, increasing the decimation rate of top voxels, and resulting in more weathering from the top.



(a) (b) (c)

Figure 3.3: (a) Original structure, (b) weathering using a spherical-shaped bubble, (c) weathering using a teardrop-shaped bubble. [21]

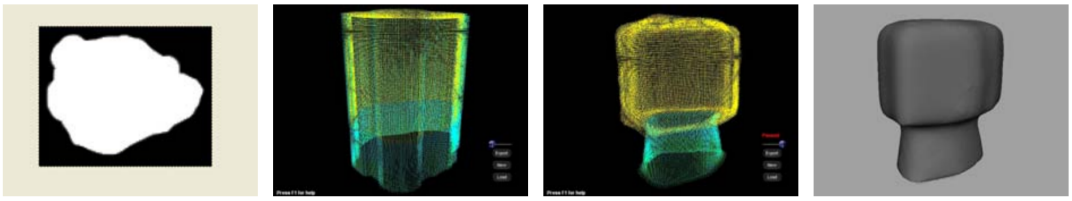
3.2.3. Proposed Method

The proposed method uses a terrain initialization step followed by an iterative procedure to weather the terrain. The method of Beardall et al. first generates a voxel structure with user-provided settings. We assume this voxel structure has already been generated and continue the explanation with the weathering procedure.

Weathering Initialization The method by [1] allows different resistance against weathering. By adjusting the weathering resistance for parts of the scene, the speed at which weathering happens can be influenced and change depending on which part of the scene is weathered. The weathering procedure starts by initializing a decimation rate for each voxel. The decimation rate d is defined as $d = (1 - w) \cdot r$, where w is the weathering resistance for that voxel (a value between 0 and 1), and r is the air-to-rock ratio in the bubble centered around that voxel. Finally, each voxel is initialized with a decimation level of 1 to indicate the degree to which a voxel has weathered.

Iterative process The weathering simulation is an iterative process that can is continued until manually terminated. In each iteration, every voxel in the scene has its decimation level reduced by its decimation rate. Because the decimation rate is proportional to the air-to-rock ratio, the decimation levels of voxels with a higher air-to-rock ratio are decreased quicker. At the end of each iteration, all voxels with a decimation level below zero are removed. To improve performance, when a voxel is removed, instead of re-calculating the decimation rate, the decimation rate of the neighboring voxels is updated and increased to account for the change in the air-to-rock ratio.

Figure 3.4 shows all steps of the weathering procedure. Figure 3.4a shows a top-down drawing used to generate the voxel terrain. Figure 3.4b shows a visualization of the different voxel properties. Figure 3.4c shows the voxel structure after the iterative weathering procedure. And finally, Figure 3.4d shows the exported 3D model.



(a) (b) (c) (d)

Figure 3.4: All steps of the weathering procedure: (a) grayscale image of the initial structure shape, (b) structure before weathering, with colors indicating different voxel properties, (c) structure after weathering, (d) exported 3D model. [1]

3.3. Arches Framework

This section outlines and discusses the most important contributions of Peytavie et al. [21]; the Arches framework, which is able to represent complex terrains including overhangs using a hybrid model. The hybrid model is a combination of a discrete data structure and an implicit representation. The discrete data structure is a volumetric data structure using material layers to perform granular simulations. The implicit representation is used for sculpting and 3D surface reconstruction.

The discrete volumetric model utilizes a layered data structure to represent the terrain. This data structure is discrete in the horizontal plane (X- and Z-plane) while continuous in the vertical axis. Voxel-based scenes are discrete in all three axes. However, the Arches framework can still be integrated with voxel-based methods, which is explained further in Section 4.3. This section focuses on the parts of the Arches framework that are relevant and used for the proposed method; in particular, the stabilization method and customization using repose angles. For the illustrations, we use 2D examples and note how these apply in 3D, if necessary.

3.3.1. Discrete volumetric data model

The discrete volumetric model utilizes material layers to create a volumetric data structure of a scene. The different material types used in the original paper are bedrock, water, air, sand, and rock. These material types are used to create material layers, defined by a material type and a layer height. Subsequently, multiple material layers on top of each other form material stacks. Arranging these material stacks in a grid results in a 3D volumetric data structure of layer stacks, as can be seen in Figure 3.5. To summarize:

- **Material type**: A type used to define what type each material layer is. The original method from Beardall et al. [1] uses the following list of types: bedrock, water, air, sand, and rock.
- **Layer**: A material layer defined by a material type and a height.
- **Stack**: A list of material layers, creating a stack of material layers on top of one another.
- **Grid**: A 2D grid of stacks to create a volumetric data structure of material layers.

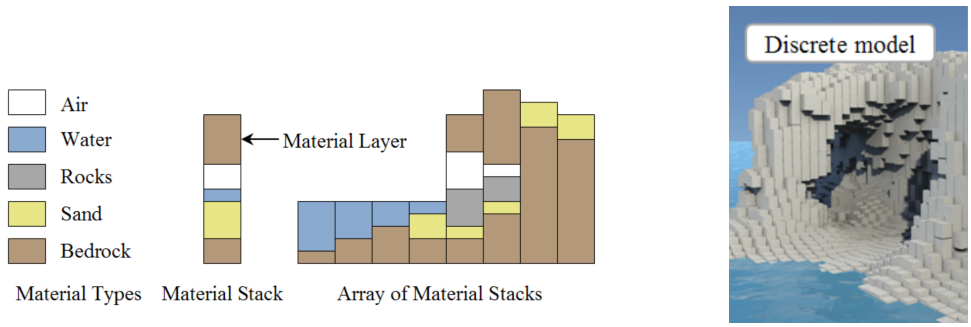


Figure 3.5: Overview of material types, layers, stacks, and 3D visualization of the discrete model. [21]

3.3.2. Stabilization

The method from Peytavie et al. proposes a stabilization procedure to displace granular material layers until the layers stabilize. Each granular layer has a preset repose angle. The stabilization procedure simulates the displacement of material layers such that these layers stabilize under this repose angle. For reference, in the original paper, the repose angle is around 30 - 35 degrees for sand and 40 - 45 degrees for rocks. The stabilization procedure is an iterative process that consists of three steps:

1. Merge stacks by material type.
2. Sort stacks by material type.
3. Displace material.

Material displacement only occurs on the granular material types sand and rock. The other material types are of type bedrock or air. Bedrock does not move and air layers are used to fill empty regions between layers.

Step 1 - Merging To simplify the stabilization procedure, the authors chose to merge all granular layers between bedrock layers. Granular layers that are interweaved or stacked are merged together into a sand and rock layer. An example of merging sand layers can be seen in Figure 3.6. While this prevents modeling different layers of sand and rock in between each other, it simplifies the stabilization procedure.

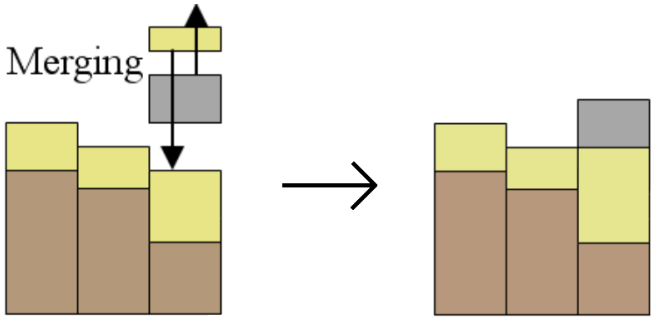


Figure 3.6: Before and after visualizations of the merging step. Material types: Brown = bedrock, yellow = sand, and gray = rock.

Step 2 - Layer sorting The authors decided to sort the layers such that rock layers are always on top of sand layers. Sorting is only applied on layers in between bedrock layers, ensuring that the bedrock layers do not move. Figure 3.7 illustrates this step.

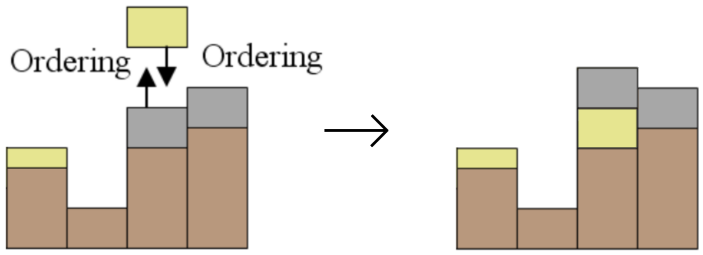


Figure 3.7: Before and after visualizations of the sorting step. Material types: Brown = bedrock, yellow = sand, and gray = rock.

Step 3 - Material displacement The last step of the stabilization method is material displacement. This step is responsible for displacing granular material such that the material slides down and stabilizes. It uses the repose angle to calculate if the material is able to fall down and ensures that material is displaced proportionally to how much material can fall to each neighboring stack. Material displacement is performed for each granular stack from bottom to top.

To determine what amount of material can be transferred to neighbors, the height difference between neighboring stacks is calculated. Let the height of the top of the center stack be h. For this 2D example, we use h_i with $i = \{-1, 1\}$. In 3D, the original method, $i \in [1, 8]$ denotes the height of its 8 neighboring stacks. For each neighbor, we determine $\Delta h_i = h_i - h$. An illustration of these heights can be found in Figure 3.8.

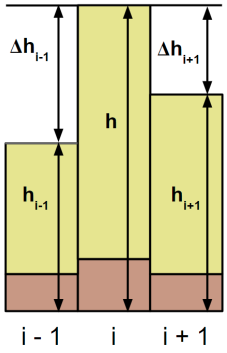


Figure 3.8: Material displacement layer heights.

The next step is to calculate how much material can fall down to neighboring stacks. Let the repose angle of a particular material stack be α and let s be the width of a material stack. We can calculate the repose height down from the top of the layer using $s \cdot tan(\alpha)$. Next, we can calculate the distance to the next layer downward from this repose height, which is used to proportionally displace material based on how much material can fall to each neighbor. For each neighboring stack i this is defined as $\Delta\tilde{h}_i$ where $\Delta\tilde{h}_i = \Delta h_i - s \cdot tan(\alpha)$. To stop material transfer when the repose angle is reached, if $\Delta\tilde{h}_i < 0$, $\Delta\tilde{h}_i$ is set to 0 to not transfer material to the neighboring stack. A visual representation can be found in Figure 3.9.

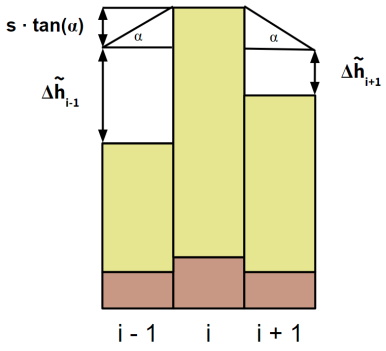


Figure 3.9: Repose angle calculations and heights.

Next, the material is proportionally displaced, meaning that more material is deposited to the neighbors that more material can fall towards. To proportionally displace material the amount of material to displace to some neighbor i is calculated and defined by Δz_i. Additionally, a constant a is introduced to avoid oscillations in material displacement, where a is a small constant value. We use $j \in \{-1,1\}$ in this 2D example, this is $j \in [1,8]$ in 3D. To proportionally displace material to each neighbor i, the method makes use of the following formula for Δz_i:

$$\Delta z_i = a \frac{\Delta \tilde{h}_i}{\sum_{j \in \{-1,1\}} \Delta \tilde{h}_j}$$

In Figure 3.10, an example is shown to illustrate the proportional displacement of material. Note that the proportions of this figure can be slightly off depending on the repose angle used in practice, however, this figure mainly illustrates the difference in material moved to either stack based on the height differences. As can be seen in Figure 3.9, the value of $\Delta \tilde{h}_{i-1}$ is greater than $\Delta \tilde{h}_{i+1}$, therefore more material is displaced to $i-1$ than $i+1$, shown in Figure 3.10.

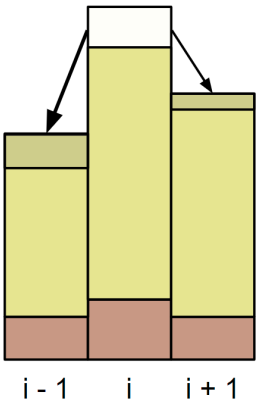


Figure 3.10: Proportional displacement of material from center material stack.

Figure 3.11 shows an example where the right stack is exactly at the bottom of the repose height. This implies that $h_i - s \cdot tan(\alpha) = \Delta h_{i+1}$, leading to $\Delta \tilde{h}_{i+1} = \Delta h_{i+1} - s \cdot tan(\alpha) = 0$. Then, as the material is proportionally distributed, no material moves from the middle stack (i) to the right stack $(i+1)$. The left stack $(i-1)$ is low enough that $\Delta h_{i-1} > s \cdot tan(\alpha)$. Therefore $\Delta \tilde{h}_{i-1} = \Delta h_{i-1} - s \cdot tan(\alpha) > 0$, shown in the green area. Since $\Delta \tilde{h}_{i-1} > 0$ and $\Delta \tilde{h}_{i+1} = 0$, the material from the middle stack (i) is able to move to the left, but not to the right.

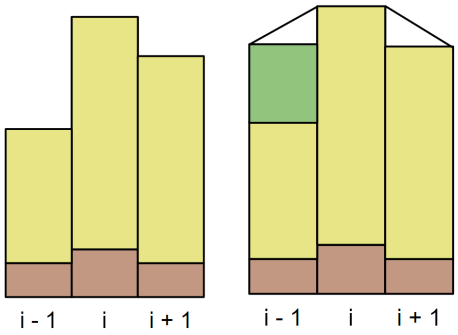


Figure 3.11: Repose angle calculations and heights.

This behavior causes the material to stabilize under the pre-specified repose angle. Users can tweak this angle to their preference.

3.4. NVIDIA CUDA

The proposed Spheroidal Weathering method of this thesis makes use of NVIDIA CUDA. This subsection provides background information on this framework.

3.4.1. Why NVIDIA CUDA?

The proposed Spheroidal Weathering editing tool (Section 4.2) requires a large number of parallel operations. The editing tool has to perform a large number of small tasks for voxel weathering. NVIDIA CUDA is well-suited for this purpose because of its highly parallelizable capabilities, which are elaborated on further. More details about the proposed Spheroidal Weathering editing tool can be found in Section 4.2, and more details about the implementation can be found in Section 5.1.

3.4.2. NVIDIA CUDA Details

Introduction [1] The CPU and GPU are designed with different goals in mind. The CPU is designed to execute a sequence of operations as fast as possible on a thread while allowing for limited parallel operations with a scale of multiple tens of threads, as opposed to the GPU which is designed to execute thousands of operations in parallel, although with worse single-thread performance. Figure 3.12 visualizes the difference in the distribution of chip resources on the CPU and GPU which also clearly shows this difference in the architecture. The CPU uses more resources for caching and flow control to allow for better single-thread performance with a lower number of cores, whereas the design of the GPU focuses on aiding parallel computations by spending more resources to increase the number of cores and less on control flow and cache.

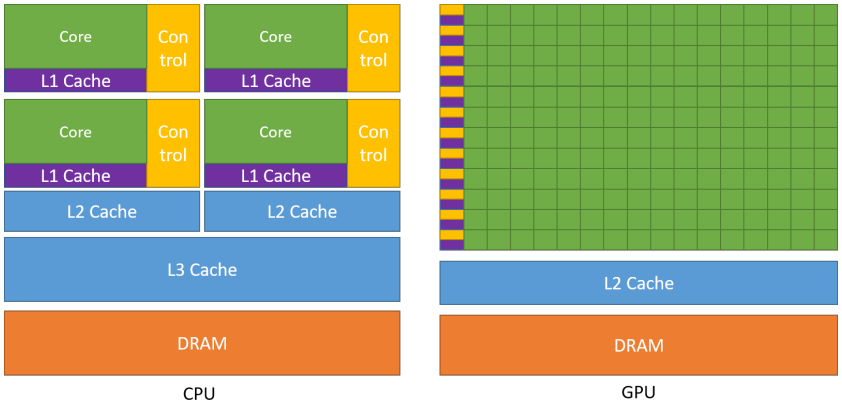


Figure 3.12: Distribution of chip resources on a CPU and GPU. [1]

[1] NVIDIA Toolkit Documentation (Introduction): `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction`

Kernels [2] NVIDIA CUDA C++ extends the C++ language by allowing developers to define a different type of function called a kernel. Kernels are specifically used for operations that utilize parallel execution by CUDA threads. These kernels are defined as a C++ function with a `__global__` declaration specifier and require a special notation that specifies the number of threads and blocks to be executed, which is covered next.

Section 5.1.2 gives insights on how CUDA kernels are used to implement the proposed Spheroidal Weathering tool.

Thread Hierarchy [3] As discussed in the previous section, kernels use threads to execute code. In this section, we discuss how these threads are orchestrated and used to execute kernels. Section 5.1.2 gives insights on how we use threads to structure and implement the proposed Spheroidal Weathering tool.

CUDA threads are organized into 1D, 2D, or 3D blocks of threads, providing common use cases such as processing arrays (1D), matrices (2D), and volumes (3D). These blocks are again organized into 1D, 2D, or 3D grids. Figure 3.13 illustrates this hierarchy of threads and blocks.

The number of blocks and the number of threads per block are to be specified when executing the kernel using a special notation provided by CUDA:
`kernelFunction<<<numBlocks, threadsPerBlock>>>(A)`, where `kernelFunction` is the kernel function, `A` is the input data, and `numBlocks` and `threadsPerBlock` are either 1D, 2D, or 3D data structures that define the size of the kernel.

The maximum number of threads for a single block is 1024 threads. Each thread is provided with variables internally to determine which part of data each individual thread should operate on. Generally, grid and block sizes are chosen such that there are enough blocks and threads to have one thread per data element.

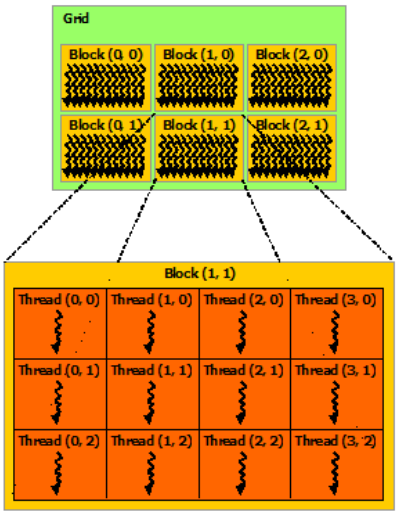


Figure 3.13: CUDA Thread Hierarchy.[3]

[2] NVIDIA Toolkit Documentation (Kernels): `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels`

[3] NVIDIA Toolkit Documentation (Thread Hierarchy): `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy`

4

Methods

This chapter discusses the methods and reasoning behind the three main proposed tools. We discuss the Spheroidal Weathering tool in Section 4.2, the Arches Simulation tool in Section 4.3, and finally the Combined Tool in Section 4.4.

4.1. Methods Overview

To preface the method sections, we give a short overview of the proposed methods. Each of these methods represents an editing tool in the HashDAG application and requires some background information to understand.

Section 4.2 discusses the proposed Spheroidal Weathering tool. It expands upon the method of Beardall et al. [1], presenting a voxel-based method to weather scenes. The proposed Spheroidal Weathering editing tool utilizes the same approach as the method by Beardall et al.. We discuss the required changes and compromises to the algorithm to make the method work in the HashDAG application. In order to understand this section fully, we recommend reading the background section on Spheroidal Weathering in Section 3.2, as this method assumes background knowledge of the methods presented in the prior work of Beardall et al.

Section 4.3 discusses the proposed Arches Simulation tool. This tool utilizes the methods from Peytavie et al. [21] and their preceding works. The integral parts are directly taken from this source: we use the material stacks and proportionally move material based on its repose angle and the height of neighboring stacks. This section describes the required changes and compromises to implement the method in the HashDAG application, on top of various changes to improve the performance of the granular simulation tool in practice. We recommend reading the background of the Arches method in Section 3.3 to understand this subsection fully.

Finally, Section 4.4 discusses the proposed combined tool. This tool uses both the proposed methods of the Spheroidal Weathering and the Arches simulation tool. This section discusses the steps to integrate both tools, extensions to the methods, and customization options. For this section, prior knowledge about both Spheroidal Weathering and the Arches method is not necessary but does give more insight into the motivation behind combining these two methods.

4.2. Spheroidal Weathering

This subsection describes the proposed method of integrating the Spheroidal Weathering methods of Beardall et al. [1] as a tool for the HashDAG framework. We cover what assumptions and changes had to be made in order to make spheroidal weathering work in real time, how the air-to-rock ratio is determined, how the weathering procedure works, and how the final voxel scene is updated in the HashDAG framework.

4.2.1. Assumption: Solid weathering area

The voxel scenes used in the HashDAG framework are all generated from 3D meshes, where voxelization is performed on the faces of the mesh. This can result in hollow structures, which can cause problems for spheroidal weathering because it uses to air-to-rock ratio to remove voxels. Figure 4.1 highlights this issue by visualizing an example of how such a voxel structure is generated for the corner of a cube. White and blue squares indicate air and solid voxels respectively, and the number represents the number of solid neighboring voxels in a 3×3 area.

Figure 4.1a shows how a voxel scene is generated for a cube. A hollow voxel structure drastically decreases the number of neighboring voxels on the boundary voxels. Because spheroidal weathering removes voxels based on higher air-to-rock ratios, this leads to the boundary voxels of the cube being removed entirely, instead of removing voxels on the edge of the cube as expected.

This problem can be solved by adjusting the voxel generation method to generate inner voxels or by filling the voxel structure before each weathering operation automatically. Solving this problem of hollow voxel structures is deemed to be outside of the scope of this thesis, and we assume that all weathering operations are performed on voxel structures where the inner area is solid. In practice, the user uses the flood fill tool from the HashDAG application to prepare the area prior to the spheroidal weathering operation. Figure 4.1b shows how this looks like for the cube structure. This shows how the number of neighbors of each voxel changes and ensures that the boundary voxels have a lower air-to-rock ratio.

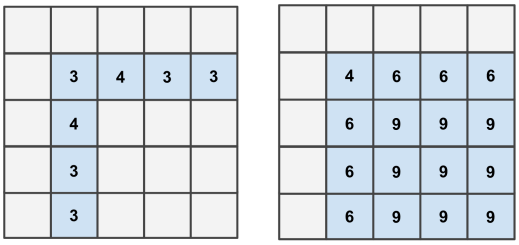


Figure 4.1: Example of hollow (a) and filled (b) voxel structure of the corner of a cube.

4.2.2. Interactive weathering

The original method makes use of the decimation rate and weathering resistance to slowly decay voxels over time. When the value of voxels reaches 0, they are removed. This process is continued until the user is satisfied. For our use case, however, we want to perform weathering interactively such that the result is presented after a single click. To this extent, we perform a single iteration and remove voxels based on the air-to-rock ratios of the voxels. Therefore, a new parameter, weathering threshold, is introduced to determine at what point a voxel is removed, based on the air-to-rock ratio in a bubble. This is similar to the use of decimation rate and weathering resistance, where the user could theoretically lower the weathering resistance so voxels are removed after 1 iteration. We introduce a weathering threshold parameter to simplify this process. Section 4.2.6 explains further details about the usage of this parameter. This allows for similar results just to the original method, with a slightly different approach using a single iteration.

4.2.3. Parameters

The first step of the spheroidal weathering tool is setting the tool parameters. All of these parameters must be set before triggering the weathering action. The following parameters are used for the weathering tool:

- **Tool region**: A cubic region defined by the selected voxel and a radius.
- **Bubble shape**: The bubble shape used for weathering iterations, which is chosen from a list of predefined bubble shapes.
- **Bubble radius**: A numeric value > 1, used to scale the bubble up and down.
- **Weathering threshold**: A numeric value between 0 and 1, defining a threshold for the air-to-rock ratio. If the air-to-rock ratio is higher than this value, the voxel is removed. This is the newly introduced parameter used to change the iterative weathering process with a single iteration for instant feedback.

4.2.4. Bubbles

As bubbles are frequently used in the explanation of further steps, we first discuss the bubble region, and introduce the weight function and bubble volume.

Bubble region The spheroidal weathering method from [1] is based on air-to-rock ratio in a local region. The concept of bubbles is used in the original method from [1] to model these local regions. To concretize these bubbles, we introduce the concept of a bubble region to define which neighboring voxels to iterate through to calculate this air-to-rock ratio.

The bubble region is defined by a 3D vector to create an Axis-Aligned Bounding-Box (AABB). To illustrate, Figure 4.2 shows a 2D example of how the vector translates to the bubble region. This directly transfers to 3D with a 3D vector.

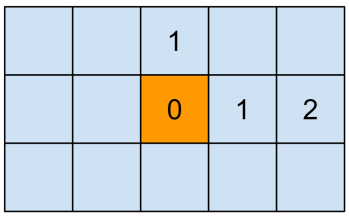


Figure 4.2: 2D illustration of a bubble region of using the 2D vector `[2, 1]`.

Weight function To create non-uniform bubbles shapes, we introduce a weight function to map voxels inside the AABB to a value between 0 and 1 based on their relative position to the center voxel. We chose a value between 0 and 1 as opposed to a binary value to allow for more precision when creating bubble shapes.

Using a constant weight of distance-based weight function can result in a cube-shaped or spherical bubble, respectively. A concrete example of a weight function is a rhombus-shaped bubble. We denote the weight function as $\omega(d) \rightarrow [0,1]$ where d is a vector for the relative position to the center voxel, and we use the variable R as the bubble radius:

$$\omega(d) = \begin{cases} 1 & \text{if } |d.x| + |d.y| \leq R \\ 0 & \text{otherwise} \end{cases}$$

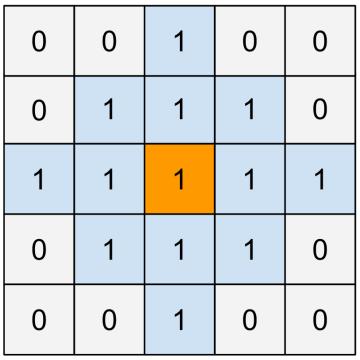


Figure 4.3: A 2D illustration of bubble weights with $R = 2$ for a rhombus-shaped bubble. The values inside each cell correspond to the weights at each cell.

Bubble volume Some bubbles account for more neighboring voxels than others while using the same size, depending on the weights. Therefore, we define a bubble volume based on the weight function to account for different shapes of bubbles. The bubble volume is defined by the sum of all values of the weight function for all voxels inside the bubble region. For example, the bubble in Figure 4.3 has a bubble volume of 13 while the bubble region contains 25 voxels.

4.2.5. Counting and Weighing Neighbors

To calculate the air-to-rock ratio at local regions, we use the bubble weight function to accumulate values of neighboring voxels. After which, we divide by the total bubble volume to get the air-to-rock ratio in the bubble. This subsection describes how the neighbors of voxels are accumulated and weighed using the weight function.

Counting and weighing neighbors Usually, a gather approach is used to accumulate neighboring voxels. However, in our case, read operations for voxels are done on a tree structure (the HashDAG), and write operations a buffer. Reading requires the traversal of the tree, which is costly. However, writing values to a buffer is much cheaper because we can directly write to the correct index using the voxel coordinates. Therefore, we decide to use use a scatter approach to scatter the weighted value of a voxel to the neighboring voxels. The value we scatter is 0 if the voxel is empty, otherwise, we add the value of the result of the weight function at that position.

Specifically, each voxel i is checked to see if it is empty, we terminate if it is empty. Otherwise, we scatter the value of voxel i to all voxels within the bubble region centered around voxel i, incrementing the buffer values using the weight function. To do this, we iterate through all the voxel positions in the bubble centered around voxel i. For some voxel j within the bubble, we first check if this voxel is inside the tool region. If it is, we use their coordinates C_i and C_j to calculate the weight $\omega(C_i - C_j)$ and increment the buffer value at the index for voxel j with this weight. We present some short discussions on early termination and thread synchronization in Section 7.1.

Extended tool region Because we use a scatter approach, if we only iterate voxels inside the tool region, voxels outside the tool region do not scatter to the voxels inside the tool region. This leads to incorrect air-to-rock ratios for voxels on the outside of the tool region. Therefore, we extend the tool region to ensure that voxels just outside the tool region scatter their values inside the tool region. We increase the tool region by half of the bubble region on each side. Extending the tool region by half of the bubble region ensures that all voxels just outside the tool region correctly scatter values inside the tool region. An example of the extended tool region is shown in Figure 4.4. An example of how voxels are counted and weighed when using this extended region is shown step-by-step in Figure 4.5.

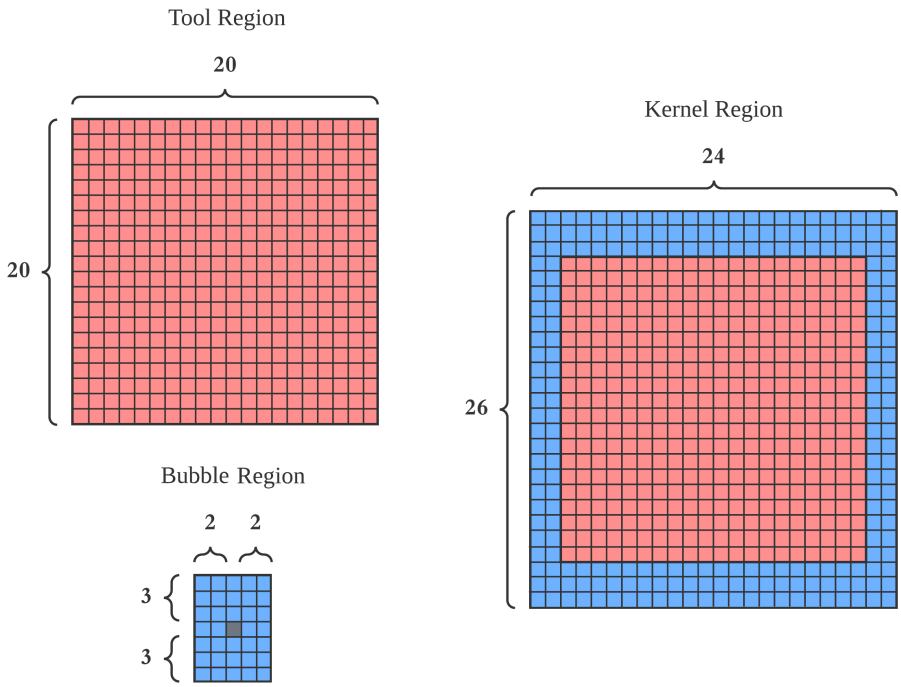


Figure 4.4: 2D Visualization of kernel size using a tool region of $T = (20, 20)$ and a bubble region of $R = (2, 3)$, resulting in a kernel size of $K = (24, 26)$.

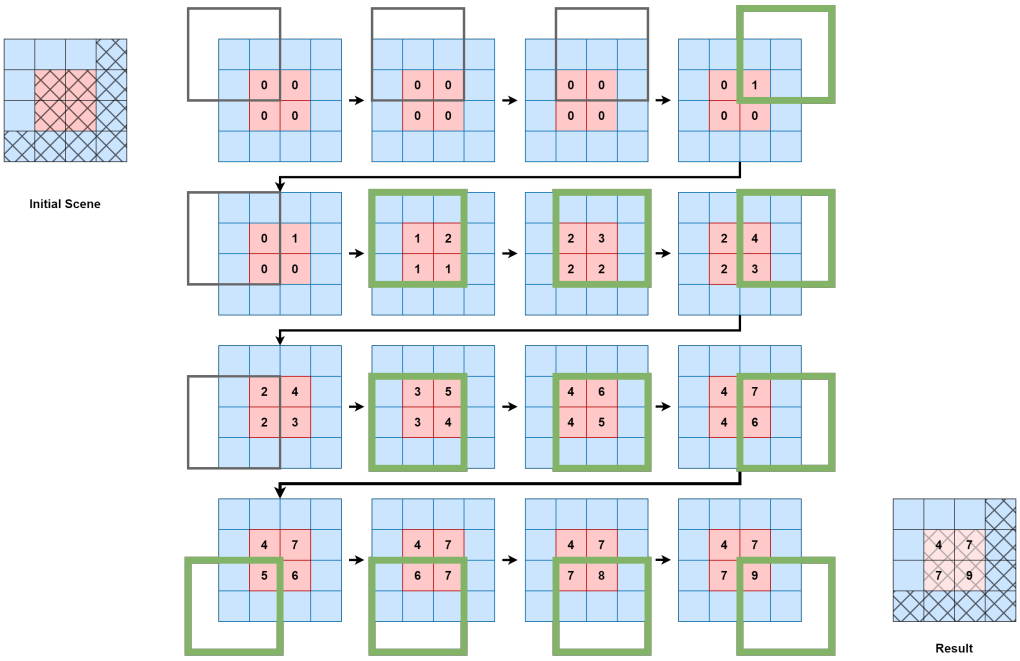


Figure 4.5: A 2D example of intermediary states of accumulating and weighing neighbor voxels. A 4×4 scene with a 2×2 tool region in the center. Crossed-out cells indicate solid voxels, others are empty. A cube bubble of 3×3 with a weight function of $\omega(d) = 1$ is used. The red cells indicate the tool region, blue indicates the extended tool region. Each step shows how values from the weight function are scattered to neighboring cells in the bubble. For each step, a green square indicates that the voxel was solid and neighboring values were incremented, black square means the voxel was empty and was skipped. The method iterates over each voxel, incrementing neighboring values. In the end, the result is a buffer where each value contains the accumulation of weights with the bubble centered around that region.

4.2.6. Removing voxels

The final step of the proposed Spheroidal Weathering tool is to determine which voxels to remove, based on the air-to-rock ratio inside the bubble for each voxel.

Bubble volume and weathering threshold As bubbles have different volumes depending on their weight function, we calculate an absolute threshold value to remove voxels. This threshold value T is defined as $T = V \cdot t$, where V is the bubble volume and t is the weathering threshold (input parameter between 0 and 1). Let's take c_i to be the sum of all weights in the bubble around voxel i. If $c_i < T$, voxel i is removed from the scene. The result of using this threshold gives comparable results to the original method. The weathering threshold parameter allows for control over how much and quickly terrain is removed.

Weathering example We illustrate the weathering and threshold calculations using an example. As spheroidal weathering should weather sharp edges and corners, we expect that parts of the corners are weathered.

A 3×3 bubble with a weight function of $\omega(d) = 1$ is used. The volume of this 2D bubble is $V = (2 \cdot 3 + 1)^2 = 49$. For this example we use a weathering threshold of $t = 0.55$, leading to an absolute threshold of $T = V \cdot t = 49 \cdot 0.55 = 26.95$. Figure 4.6 shows a 2D example of a corner of a square of voxels. This square further extends in the downward and right direction outside of the figure. All voxels with a value below $T = 26.95$ are weathered and colored red. As expected, these are the voxels on the corner of the cube. Tweaking the bubble radius, weight function, or weathering threshold leads to different results.

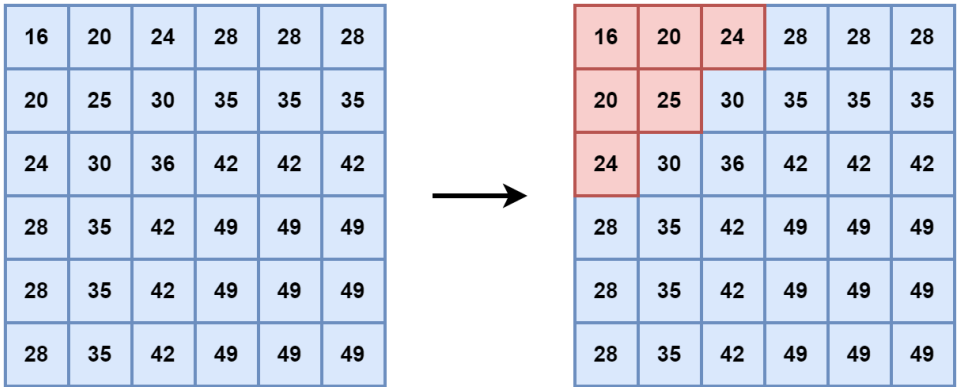


Figure 4.6: Visualization of accumulating and weighing neighbors using a cube-shaped bubble with a radius of 2: (1) showing all weighed nearby neighbors for all voxels, and (2) highlighting the weathered voxels.

4.3. Arches Simulation

This section describes the proposed editing tool for granular simulation, utilizing the methods of the earlier work of Peytavie et al. [21] to create a granular simulation tool for the HashDAG framework. This section covers how this tool is used in the application, the three different steps of the Arches simulation method, and what changes were required to make it work in the HashDAG application. The Arches steps are (1) stack initialization, (2) simulation, and (3) updating the voxel scene with new simulated granular material.

4.3.1. Usage in HashDAG Application

Before discussing the details of the proposed simulation tool, we explain some details about using the granular tool in the HashDAG application. This includes the flow of the tool, input parameters, and the concept of a partial simulation region.

Tool flow In the HashDAG application, tools are implemented such that clicking a voxel initiates an edit operation, and control is given back to a user once the operation has been completed. Therefore, the proposed granular simulation tool functions in the same manner. The user clicks a voxel, the granular simulation edit tool is initialized, granular material is simulated, the HashDAG scene is updated with newly added voxels from the granular simulation, and after that, the user regains control of the application.

Input parameters The simulation tool adds granular material in a region and simulates this newly added material. The input parameters required to perform granular simulation are a tool region and a material type:

1. **Tool region**: A cubic region defined by the clicked voxel and a radius.
2. **Granular material type**: A material type defined by a repose angle for which the material stabilizes, a color that defines what color to assign to the added voxels and a name. A material type is chosen from a predefined list of types.

4.3.2. Flexible Simulation Region

The HashDAG application [5] enables real-time editing of high-resolution voxel scenes, allowing sizes up to $2^{17 \cdot 3}$ voxels. To use the Arches simulation method, we need to generate material stacks in the region we want to simulate. If we generate material layers for each voxel in the scene, a 2D grid of $2^{17 \cdot 2}$ stacks are required. This amount of data is infeasible to store without a novel compression method for layered data structures. Additionally, if we use a data structure that spans the complete scene, we run into issues when performing other editing operations in the HashDAG application. Each edit operation unrelated to the Arches simulation needs to update the layered Arches data structure. This adds major overhead to each edit operation.

For those two reasons, we propose only initializing layers inside the tool region. During simulation, we do allow material to fall outside the tool region. When this happens, new layers are generated for uninitialized parts.

4.3.3. Material Stack Generation

Material stacks are generated when initializing the tool and whenever material falls into uninitialized regions. Stack generation converts the voxels in the HashDAG scene into material layers so granular material can be simulated. The layer generation is slightly different when generating layers during simulation as opposed to during initialization, we cover both. These illustrations are in 2D, but the methods work the same in 3D.

Initialize new stacks during simulation When using the Arches granular simulation tool on a voxel scene, stacks need to be generated from a voxel scene. This generation is performed for each stack in the tool region, we focus on generating a single stack. Figure 4.7 shows this generation procedure for the left. The stack generation starts at the voxel at the respective grid position and some specified height (when material falls to the side, this refers to the height of the original material stack). Next, we iterate downward through the voxels, starting at the specified height. For each voxel, if the voxel is empty, it is converted into an air layer or merged with previous air layers. Otherwise, if the voxel is solid, a solid layer is generated and the stack generation method terminates. Finally, when the method terminates a base height layer is added below the solid layer with the height of the Y-position of that voxel to ensure that the absolute height of the layers matches that of the voxels.

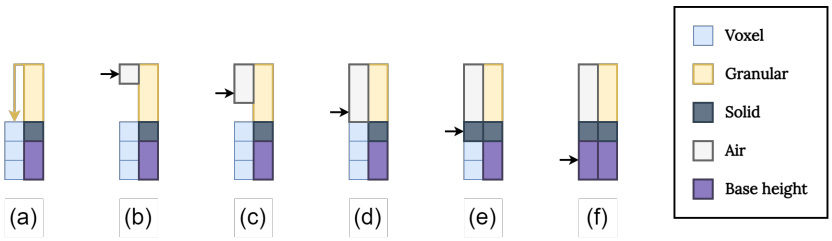


Figure 4.7: Generation of new stacks in Arches simulation method.

Initialize stacks in tool region When initializing the tool, material stacks in the tool region are to be initialized with granular material instead of air. Therefore, the stack generation method we just explained is slightly adjusted to generate stacks in two steps:

The first step is to generate the stacks for the voxels inside the tool region. However, now instead of terminating when a solid voxel is hit, we continue to the bottom of the tool region. And for empty voxels, instead of generating air layers, we generate granular layers.

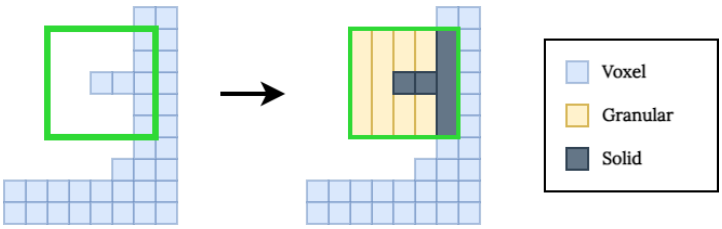


Figure 4.8: 2D visualization of partial initialization inside of a tool region.

The second step is to generate stacks for the voxels down from the tool region so the material has something to fall onto. This part functions the same as generating layers during simulation: empty voxels are converted to air layers, we terminate when a solid voxel is hit, and a base height layer is generated below the first solid voxel.

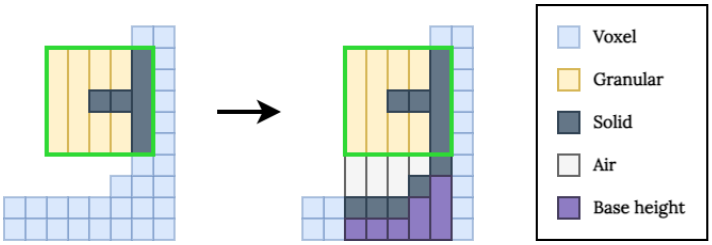


Figure 4.9: 2D visualization of partial initialization below the tool region.

Overhangs The Arches simulation allows for the simulation of interweaved solid, air, and granular layers to simulate overhangs. With our layer generation method, we only generate materials for a partial region in the scene, from some height down to the first voxel. Above and below that region are not generated to speed up the layer generation method. Inside the tool region, layers are generated for each voxel, so overhangs are correctly generated and simulated inside the tool region. More details about this limitation can be found in Section 8.2, and suggestions for future work in Section 10.2.

4.3.4. Step 1 - Initialize simulation

The user initiates the Arches simulation tool using a tool region (a cubic region around some voxel). The proposed Arches method makes use of a partial simulation region, as discussed in Section 4.3.2. The first step is to initialize the layers inside the tool region. We generate stacks for each 2D position in the tool region, downward from the top of the tool region, as explained in Section 4.3.3.

4.3.5. Step 2 - Simulation

The simulation method of the original Arches method [21] has been slightly adjusted to work in the HashDAG application. First, we cover why we decided to restrict the simulation to the tool region. Then, how we handle stacks falling outside the initialized region during simulation. And finally, we cover an optimization with active stacks to skip simulating stabilized stacks.

Remove diagonal material movement In the original method, stacks transfer material to its 8 neighbors. We perform the simulation by only transferring to the 4 direct neighbors because of the performance benefit of only transferring to the 4 neighbors instead of 8.

Additionally, the original method of transferring to 8 neighbors does not account for diagonal walls. Figure 4.10 illustrates the difference between using 8 (Figure 4.10a) and 4 neighbors (Figure 4.10b). Figure 4.10a highlights the issue when using 8 neighbors, the material flows to the top-right through the wall. In our case, the material stacks are generated from the HashDAG voxel scene, which is a voxelization of faces from a 3D mesh. Therefore, examples like these diagonal walls occur much more frequently because most surfaces are one voxel thick. And because material tends to flow through these diagonal wall sections, we consider 4 (horizontal and vertical) neighbors instead of 8 (horizontal, vertical, and diagonal).

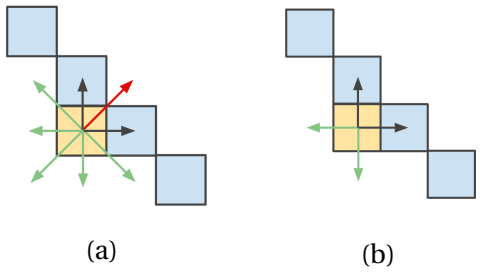


Figure 4.10: Top-down view of a granular material layer alongside a diagonal wall of solid layers. Visualization of using 8 neighbors (a) compared to 4 (b) neighbors during the simulation.

Initialize stacks during simulation Stacks outside of the tool region are not generated at the time that the simulation starts. If a stack is to deposit material into one of these uninitialized stacks, this stack height is unknown. In this case, we generate this new stack from the height of this material stack downward. How this generation method works is discussed in Section 4.3.3. After generating the stack, it is updated in the grid, and we continue the simulation with the newly updated grid.

Active stacks When using a partial generation region is that it is unknown which stacks contain granular material, and which stacks require simulation. Therefore, we introduce the concept of 'active' stacks. Active stacks are a set of pointers to stacks in the grid containing granular material that has not yet stabilized. During simulation, we can iterate over the set of active stacks and perform simulations for each stack. This allows us to keep track of all stacks that still require simulation and avoid simulating stacks that do not require simulation. Because we use pointers to material stacks, this also saves a lot of time fetching stacks from the grid.

We initialize the active stacks to all stacks containing granular material. During the simulation, we add and remove stacks from the set of active stacks. If a stack stabilizes (it does not deposit any material) we remove the stack from the active stack list. If a stack deposits material, the stack stays in the active stack set. Additionally, if a stack deposits material, it adds all the neighboring stacks around it to the active stacks set to indicate that these stacks may now able to deposit material to the current stack.

Figure 4.11 illustrates an example of active stacks in practice by starting with a region of granular material. Initially, all stacks containing granular material are active, indicated by their green color in Figure 4.11(a). The first iteration is shown in Figure 4.11(a) where only stack 4 deposits material to the right. Stack 4 stays active and activates the neighboring stacks 3 and 5. Stacks 1 and 2 do not deposit material. These are stabilized and removed from the active stacks, indicated in red. Figure 4.11(b) shows the next iteration where stacks 3, 4, and 5 deposit material to the right. This keeps stacks 3, 4, and 5 active and activates neighboring stacks 2 and 6. In the next iteration in Figure 4.11(c), no stacks can deposit material anymore. This stabilizes all stacks and removes all stacks from the active stacks. Figure 4.11(d) shows the next iteration where no active stacks are left, terminating the simulation procedure.

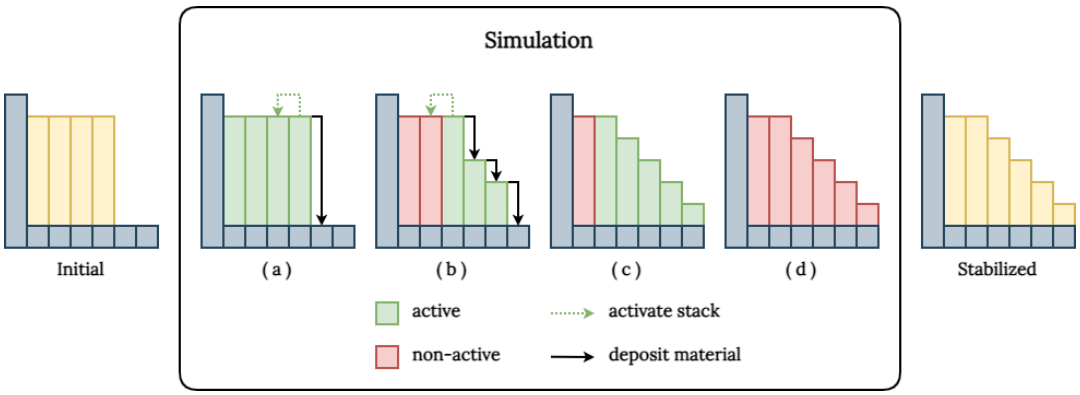


Figure 4.11: 2D visualization of adding and removing active stacks during simulation.

4.3.6. Step 3 - Update voxel scene

The final step of the Arches simulation tool is adding the granular material layers as voxels in the HashDAG framework.

Update boundary Editing tools in the HashDAG application require a bounding box to indicate what voxels in the scene should be updated. As noted before, granular boundaries are introduced and updated during the simulation, these are used here. When updating the voxels in the scene, only the stacks containing granular material are relevant, because the granular simulation tool only adds material. It does not affect the original scene.

Material layers to voxels To update the voxel scene, material layers should be transformed into voxels. Voxels always have a height of 1, while material layers can have any fractional height. This can lead to cases where material layers do not span a voxel completely. One solution for this problem is to take the layer type that has the majority of the height in the space in a voxel. This solution results in the most accurate representation of the layers but requires more bookkeeping to accumulate the material distribution in each voxel in this final step, while we can take a simpler approach.

Because solid layers are generated from voxels, solid layers always have a height of 1. Additionally, with the Arches simulation tool, we only use a single granular layer type at a time. Therefore, only the granular layers have fractional heights. So we can use the layer type at the midpoint of a voxel to determine what material type to set for a voxel. This yields the same result as using the voxel that spans the majority of a voxel. Figure 4.12 illustrates how this converts material layers into voxels.

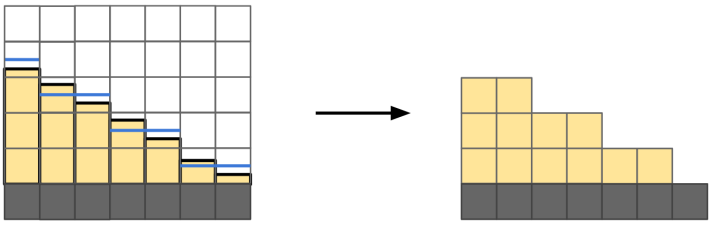


Figure 4.12: Determine voxel material type using the material in the middle of a voxel.

4.4. Combined Tool

This subsection discusses the combined tool, a combination of the Spheroidal Weathering and Arches simulation methods.

4.4.1. Motivation

The spheroidal weathering method of Beardall et al. [1] removes weathered material from the scene, like other similar works as discussed in Chapter 2. In reality, removed material is either blown away or falls to the ground. The proposed granular simulation tool of this thesis (discussed in Section 4.3) can be used as an extension to simulate debris material. The Spheroidal Weathering method is performed on voxels, and the Arches simulation method makes use of a 2D grid with vertical material layers. This allows us to integrate both methods easily by representing the removed voxels as material layers of height 1 and use of the Arches simulation method to simulate the weathered voxels.

4.4.2. Method Steps

This subsection describes the steps of combining the Arches and Spheroidal Weathering methods to create a combined tool to simulate the debris of removed material. Figure 4.13 shows a 2D illustration to explain the steps. The same steps apply in 3D, where the Arches and Spheroidal Weathering methods are also used in 3D.

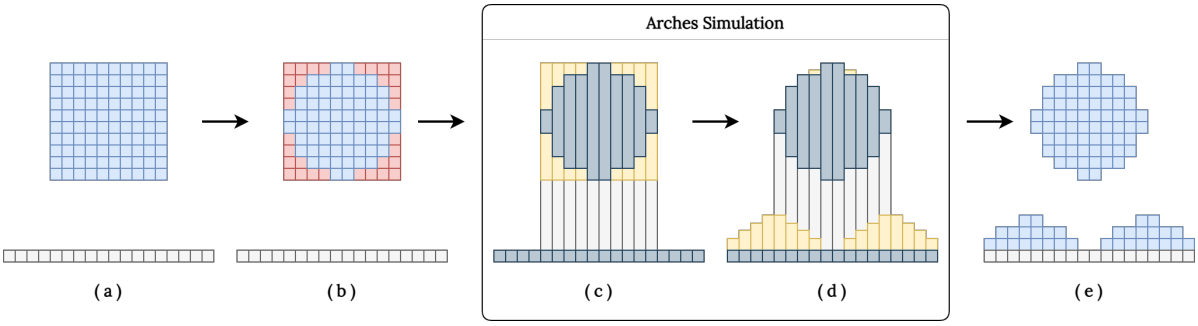


Figure 4.13: 2D illustration of the combined tool steps. Step (a) shows the initial voxel scene. Step (b) shows the result of weathering with the voxels to remove shown in red. Step (c) shows the initialized material layers based on what voxels to remove. Dark blue layers are solid layers, yellow layers are granular layers, and white layers are air. Step (d) shows the result after simulation the Arches simulation. Finally, step (e) shows the result of converting the Arches layers into voxel, also retaining their color.

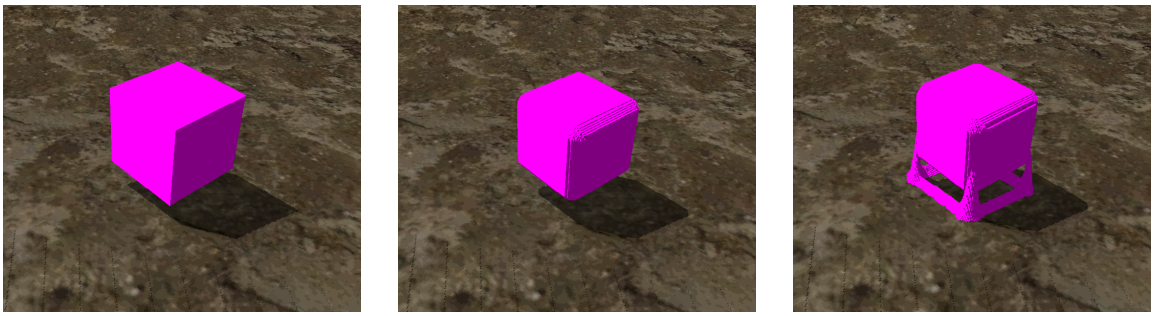
Step 1 - Spheroidal Weathering First, spheroidal weathering is performed as described in Section 4.2. In the application, you select the weathering parameters and click a voxel in the scene. For this example, the tool region surrounds the complete cube. This is shown in Figure 4.13(b).

Step 2 - Initialize Arches material layers This step is illustrated in Figure 4.13(c). We use the removed voxels to initialize layers for the Arches simulation. Removed voxels by spheroidal weathering are added as granular layers. All remaining solid voxels are added as solid layers. All other empty voxels are generated as air layers. And finally, we generate layers downward from the tool region to the first solid voxels, as explained in Section 4.3.4. This results in the layers in Figure 4.13(c).

Step 3 - Simulate granular material After initializing the layered data structure, the Arches simulation method is performed as specified in Section 4.3.5. The material is simulated until the material stabilizes or until some maximum number of iterations is reached. This simulates the debris material falling down off the scene structure. The result of this step can be seen in Figure 4.13(d).

Step 4 - Update HashDAG voxel scene The final step is to add the simulated debris material as voxels after the granular simulation is finished. This is done using the same method as the Arches tool, as specified in Section 4.3.6. This step only adds the debris material as voxels, because we have already removed the material in Step 1. The result is visualized in Figure 4.13(e). How we assign colors to the debris voxels is explained in Section 4.4.3.

3D example The previous illustrations were all using 2D examples, in practice, the tool is utilized on a voxel scene in 3D space. Figure 4.14 illustrates the steps of this tool using a 3D example. Figure 4.14(a) shows the initial scene; a cube of voxels. Figure 4.14(b) shows this cube after weathering, corresponding to Figure 4.13(b) of the 2D example. Then Figure 4.14(c) shows the 3D scene after debris voxels are simulated and added back to the voxel scene, corresponding to Figure 4.13(e) of the 2D example.



(a) (b) (c)

Figure 4.14: Visualization of the intermediate steps of the combined tool. (a) The initial scene of a voxel cube. (b) The cube after weathering. (c) The weathered cube after simulating and adding the debris material.

4.4.3. Debris color

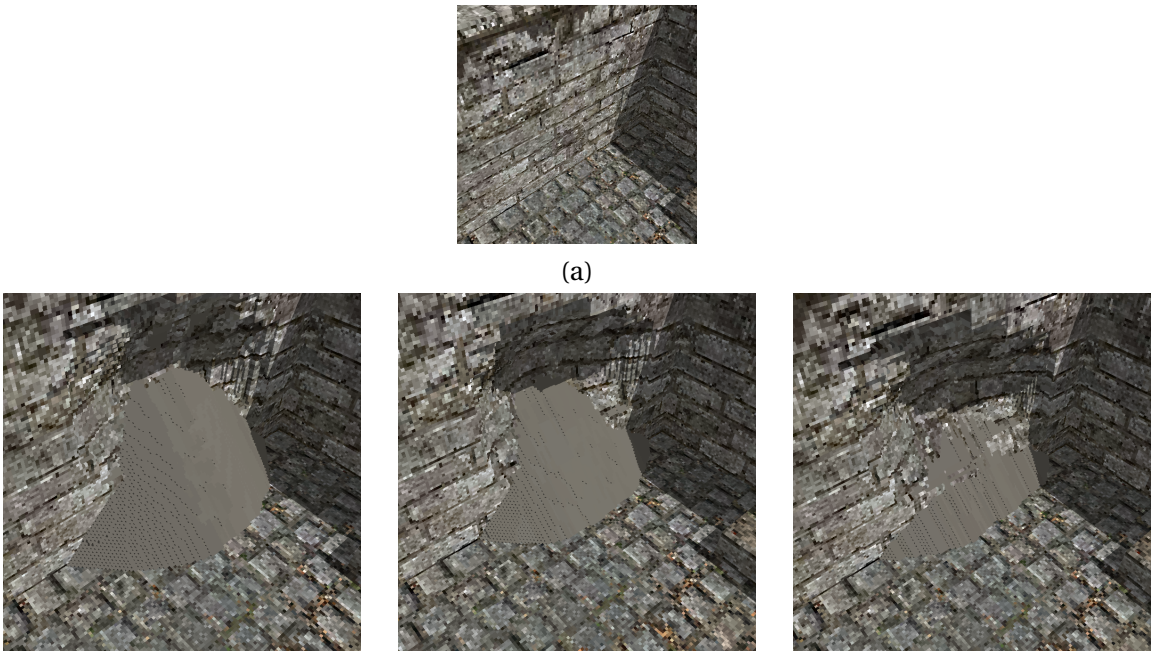
One detail about adding the debris material to the voxel scene is the debris color. Ideally, the removed voxels carry over their original color to the debris voxels. However, the Arches method groups voxels into material layers, making it difficult to keep individual voxel colors. To solve this, we average the color of all removed voxels and use that for all debris voxels. This works well to carry over the color of regions with similar colors. The average color is less accurate for regions with many varying colors. However, we deem this problem to be out of the scope of this thesis and propose this as future work with suggestions in Section 10.3.

4.4.4. Customization

To give the user more control of the debris simulation, two customization parameters are added: debris percentage, and debris material type. This allows the user to customize the debris simulation behavior and amount of debris.

Debris percentage A debris percentage parameter is introduced because converting all weathered material to debris material stacks can cause a lot of debris to pile up. This parameter reduces the number of removed voxels that are converted into debris material. It is used in Step 2, where the Arches material layers are generated. Each removed voxel is randomly converted into either debris or air, based on the debris percentage. The percentage indicates the chance of converting a removed voxel into debris.

Figure 4.15 illustrates the impact of the debris percentage parameter. Figure 4.15(a) shows the initial scene, a solid vertical wall. Figure 4.15(b) shows the wall after being weathered with 100% debris material. Figure 4.15(c) uses 50% debris. And Figure 4.15(d) uses 20% debris. We can clearly see the impact of the debris percentage parameter, the wall is weathered to the same degree, but less debris material is deposited into the scene.

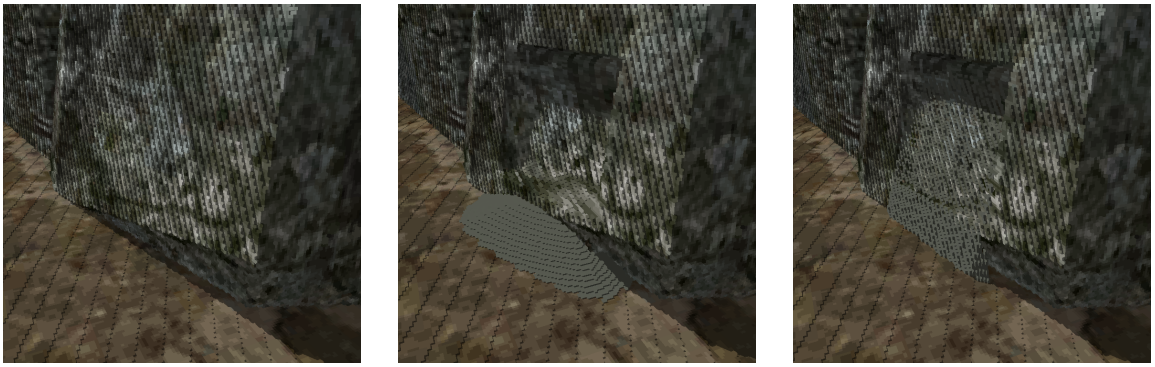


(a) (b) (c) (d)

Figure 4.15: Examples of a different debris percentages. (a) Before weathering, (b) 100% debris percentage, (c) 50% debris percentage, and (d) 20% debris percentage.

Debris material type The combined tool makes use of the Arches simulation method, which enables different material types with different repose angles. The second parameter for the combined tool is selecting the granular material type to use for debris simulation. Like the Arches tool, this is also from a list of predefined material types. This material type is used to initialize the granular layers of debris material.

Figure 4.16 shows the result of the combined tool of 15 edit operations with 50% debris with different debris materials. Figure 4.16(a) shows the scene before any weathering, a sloped wall of solid rock. Figure 4.16(b) uses a material with a repose angle of 10 degrees. 4.16(c) uses a material with a repose angle of 75 degrees. We can clearly see the impact of the repose angle, the stabilized material in Figure 4.16(b) is much flatter than the material in Figure 4.16(c).



(a) (b) (c)

Figure 4.16: Example of different material types using the combined tool. (a) Before weathering. (b) Weathering using a material with a repose angle of 10 degrees. (c) Weathering using a material with a repose angle of 75 degrees.

5
Implementation

This section discusses the implementation details of two of the proposed tools. Section 5.1 discusses the Spheroidal Weathering tool and Section 5.2 discusses the Arches Simulation tool.

5.1. Spheroidal Weathering

This subsection discusses the implementation details of the Spheroidal Weathering tool. In particular, NVIDIA CUDA and the data flow for each of the steps of the editing tool.

5.1.1. NVIDIA CUDA

NVIDIA CUDA (in short, CUDA) allows processes to be parallelized on the GPU, significantly improving performance for parallelizable tasks. For more details and background information about NVIDIA CUDA, see Section 3.4. The spheroidal weathering method utilizes bubbles to accumulate and weigh neighboring voxels around each voxel. This process is well suited for parallelization because the same operation is performed for each voxel. Therefore, we chose to use CUDA to leverage the parallel processing power of the GPU to improve the performance of the spheroidal weathering tool.

Additionally, because the HashDAG application already makes use of CUDA to store the voxel scene, the implementation of the CUDA optimizations of the spheroidal weathering scene is much easier.

5.1.2. CUDA steps

The weathering procedure implemented with CUDA consists of three different steps:

1. Initialize buffer for 3D grid on GPU and CPU
2. Execute CUDA kernel
3. Collect results from CUDA kernel

Step 1 - Buffer initialization The first step for the CUDA part of the Spheroidal Weathering tool is preparing the data buffer for the GPU. The spheroidal weathering tool uses a cubic tool region of `N = size`^{3} voxels, where `size` is the length of one side of the cuboid tool region. The data type of the buffer is a `float` because neighbors are accumulated and weighed using the weight function that returns a `float` value. The buffer is initialized using both `cudaMalloc` and `cudaMemcpy`. The buffer has a size `N = size`^{3}, all default to 0. We do discuss the implications of using a buffer spanning the complete tool region in Section 8.1.

Step 2 - Execute CUDA Kernel CUDA kernels are executed in blocks of threads. We use a single thread per voxel. And we use a block size of $8 \times 8 \times 8$, so $8^3 = 512$ threads because 512 is the largest possible 3D block size with a power of 2 below 1024 (the maximum number of threads in a block[1]).

As discussed in Section 4.2.5, buffer values of nearby voxels are incremented in the accumulation and weighing process. As discussed in Section 4.2.5, this requires additional space at the edges of the kernel to account for the voxels just outside of the tool region. Therefore, the total size of the kernel is the total size of the tool region plus the size of the bubble, as shown in Figure 4.4. The total kernel size K is denoted below, with tool region T (cubic-shaped region, implies $T_x = T_y = T_z$), and with - possibly asymmetric - bubble region R:

$$K = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} + 2 \cdot \begin{bmatrix} R_x \\ R_y \\ R_z \end{bmatrix}$$

An implication of using a thread for each voxel is that some threads terminate before others. This is because we terminate the thread is terminated immediately when it fetches the voxel value and it's empty. We discuss this a bit more in Section 7.1.

Step 3 - Collect results from GPU The final step of the Spheroidal Weathering tool is copying the data from the GPU back onto the CPU after the kernel has completed execution. Using `cudaMemcpy`, all the data from the buffer is copied from the GPU to the CPU. This buffer contains the accumulation of the weighing of the neighbors around each voxel in the tool region and is now available on the CPU to perform the weathering.

[1] NVIDIA Toolkit Documentation (Thread Hierarchy): `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy`

5.2. Arches Simulation

This section describes the implementation details of the proposed Arches granular simulation tool. In particular, the data structures, the simulation, and updating the voxel scene once the simulation is completed.

5.2.1. Grid data structures

The Arches method uses a grid of stacks to simulate material. From the lowest level, this grid is made up of:

Structure	C++ Data Type
LayerType	`enum: uint8`
Layer	`std::pair<LayerType, float>`
Stack	`std::vector<Layer>`
GridEntry	`struct { Stack& stack; int x, y; std::array<std::optional<GridEntry*>, 4> neighborPtrs; int lastActiveIteration; }`
Grid	`QuadTree<GridEntry>`

Layer type Layer type is implemented using a `uint8` (8-bit unsigned integer). The basic layer types are `BaseHeight (0), Unknown (1), Solid (2), Granular (3)`. It can be easily extended to add more types.

Layer Layers are defined by a single material type and a height. They are implemented using the C++ `std::pair<LayerType, float>` data type. Using a float value for layer height allows for sub-voxel height changes in the simulation, allowing for repose angles below 45 degrees.

Stack Stacks are defined as layers 'stacked' on one another. These are implemented using the C++ standard library vector implementation: `std::vector<Layer>`. A dynamic-sized array type is chosen over a static-sized array type as layers are added and removed during simulation.

GridEntry We introduced a GridEntry object, a C++ struct that stores its position, a Stack object, and optional pointers to neighboring grid entries. These pointers are all uninitialized by default and are initialized as needed during simulation iterations. They are reused during subsequent iterations, reducing the number of fetch operations required to get the neighboring stacks. Finally, the last active iteration number is stored in the grid entry for optimization of active stacks, more details in Section 5.2.2.

Grid The Grid contains all the required data for the Arches simulation. It is a 2D grid of GridEntry objects used to simulate the granular material. The Arches simulation requires many fetches of the Grid data structure. The grid also needs to be dynamic due to the behavior of the granular material in the simulation as new stacks may need to be generated for uninitialized positions.

We decided on using a QuadTree data structure because it has relatively fast lookup speeds due to its spatial hierarchical data structure and can be extended with new material stacks during simulation.

We have also experimented with a hash map (or `std::unordered_map` in C++). We expected this to give better results due to the reduced overhead of spatial data structure. However, it gave similar results in terms of runtime. We theorize that this is because our method makes use of a list of active stack pointers and neighbor pointers for each stack, both using direct pointers to grid entries. This reduces the number of grid fetches during the simulation drastically. We theorize that this diminishes the effect of using a hashmap as opposed to a QuadTree when compared to the total runtime. Because the difference was minimal, we still use the QuadTree data structure.

5.2.2. Simulation

This subsection discusses the simulation step of the Arches simulation tool. In particular, we discuss the termination condition, how the heights of neighboring stacks during simulation are calculated, the stabilization condition (or deposit threshold), the active stacks mechanism, and the neighbor pointers optimization.

Termination The simulation continues until no active stacks are left or until some preset number of iterations is reached to avoid infinite simulations.

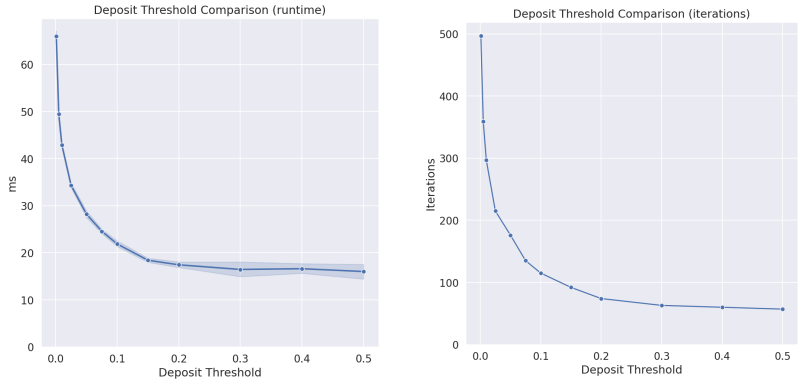
Read and writing to the grid Since material stacks deposit material based on the repose angle, when depositing material, it should be known how far down the neighboring stacks are at that moment. Usually with simulations, a ping-pong method is used with two grids, alternating one for reading and one for writing. If a ping-pong method is used, we transfer material to the second grid, while using the repose heights from the first grid. This may lead to inaccurate results with multiple stacks transferring to the same stack. In our case, we decided to implement this by reading and writing to the same grid during the simulation to ensure that we do not transfer too much material to neighboring stacks. This does have some implications, as we further discuss in Section 8.2.

Neighbor heights during simulation During the Arches simulation procedure, the height of neighboring stacks is required for a particular layer to deposit material. To determine the height, the layers are iterated upward through the stack. However, layers that are higher than the current stack are not relevant, as the material cannot fall onto stacks that are higher. Therefore, we stop iterating when the height is past the height of the layer that is depositing material.

Additionally, when a neighboring stack contains an air layer as the topmost layer, it is not used for the height calculation. This allows the material to be deposited, even when an air layer is on top. This is just an edge case, but important to account for.

Stabilization condition Float values are used in the implementation of layer heights. If there is no threshold for depositing material, it can be deposited indefinitely while approaching a layer height of 0. To avoid this, the deposit threshold value is introduced. If the amount of material is below the threshold, no material is deposited and that particular stack is deemed to be stable.

To choose a viable threshold, we experiment with different values such that the number of iterations is minimized while also minimizing the impact of the final result. Figure 5.1 shows the impact of increasing the threshold on the run time and the number of iterations until stabilized. From these graphs, we can conclude that the runtime and number of iterations until stabilization decrease much less drastically after a threshold value of 0.2. In the graph, we see that 0.1 gives a good trade-off between a low threshold and runtime/iterations. Therefore, we decide to use a threshold value of 0.1 for the simulation.



(a) (b)

Figure 5.1: Visualization of the impact of different deposit thresholds. Figure (a) shows the runtime of the Offline Arches benchmarks. Figure (b) shows the required number of iterations to stabilize.

Figure 5.2 shows the impact of the deposit threshold on the stabilized material of creating a cube of granular material with a repose angle of 45 degrees. Using a deposit threshold greater than 0.1 results in a different stabilized angle than the repose angle with rougher edges. Using an angle less than or equal to 0.1 we see that the stabilized material matches the 45-degree repose angle better. We do note that using a lower deposit threshold below 0.1 improves the accuracy of the set repose on the stabilized edges, as you can notice in the difference between 0.001 and 0.1. We do note that this is a tradeoff between quality/accuracy and performance, where we decided on performance over quality by choosing the deposit threshold of 0.1.

We have experimented with an adaptive deposit threshold based on the repose angle with reasonable results. However, more extensive work is left for future work, we discuss it more in Section 10.2.

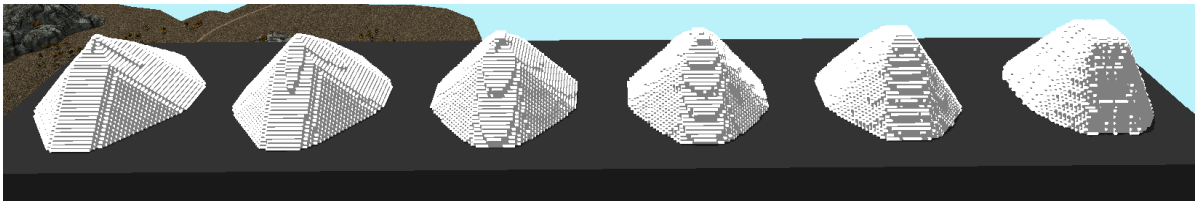


Figure 5.2: Impact of different deposit thresholds on the stabilized granular material.
(From left to right: 0.001, 0.01, 0.05, 0.1, 0.2, 0.5)

Active stacks As discussed in Section 4.3, active stacks are used to keep track of non-stabilized stacks with granular material. In each simulation iteration, we iterate over all of these active stacks to simulate all non-stabilized material.

In theory, the active stacks can be implemented using a set or HashSet (`std::unordered_set`). However, during simulation, an already active stack can be activated by some other neighbor. Checking these duplicates leads to unnecessary costly operations as the active stack set grows. Therefore, we optimize this by implementing the set using a `std::vector` (as `std::vector<GridEntry*>`) with an additional `lastActiveIteration` field for each grid entry. We set this field to the last iteration a stack was active. We can then use this value to avoid adding duplicates by only adding active stacks when the last active iteration was in a previous iteration.

We describe the process using pseudocode in Algorithm 1. For each iteration, we start with a list of active stacks and an empty list to store the new active stacks for the next iteration. We iterate through all stacks in the active stack list. For each of those stacks, we perform a simulation iteration.

If it was stabilized (no material was moved), we continue to the next stack. Otherwise, we add the current stack and neighboring stacks to the new active stack list. Each stack contains the iteration number in which it was last added to the active stacks. We update the last active iteration for the current stack and all neighboring stacks and add these to the new active stacks list. This prevents adding duplicates to the active stacks. After simulating an iteration, we set the current active stacks to the new active stacks and reset the new active stacks. We stop simulating if there are no active stacks left and all stacks have stabilized.

Algorithm 1 Simulate granular material

```
1: procedure SIMULATESTACKS(activeStacks, maxIters)        ▷ activeStacks is active stack list
2:     newActiveStacks ← []                                ▷ Initialize new active stacks to empty list
3:     for curIter ← 0 to maxIters do
4:         for s ∈ activeStacks do                         ▷ Iterate through all active stacks
5:             Perform simulation on stack s
6:             if s is stabilized then                     ▷ Continue to next stack if this stack has stabilized
7:                 continue
8:             end if
9:             if s.lastActiveIteration < curIter then     ▷ Activate current stack
10:                s.lastActiveIteration ← curIter
11:                newActiveStacks += s
12:            end if
13:            for neigh ∈ s.neighborStacks do             ▷ Activate neighboring stacks
14:                if neigh.lastActiveIteration < curIter then
15:                    neigh.lastActiveIteration ← curIter
16:                    newActiveStacks += neigh
17:                end if
18:            end for
19:        end for
20:        activeStacks ← newActiveStacks
21:        newActiveStacks ← []
22:        if activeStacks is [] then
23:            break        ▷ Stop simulation if there are no active stacks, all stacks have stabilized
24:        end if
25:    end for
26: end procedure
```

Optimization: Neighbor pointers To deposit material from one stack to the neighboring stacks, each stack needs to fetch the four neighboring stacks to determine their heights. Meaning that for each simulation iteration, for each stack, 5 fetches were required in total.

This was optimized by storing the pointers to all four neighbors for each stack, reducing the number of fetch operations. A GridEntry stores a Stack and four optional neighbor pointers, pointing to the neighbors of the respective Stack object. These pointers all start empty before the simulation. As the simulation progresses, if a neighboring stack is fetched, the pointer to that neighboring stack is stored in the GridEntry. The neighbors are then accessed directly through those pointers in further iterations of the simulation. This optimization led to a 1.99× speedup. The benchmark results of this optimization can be found in Figure 6.18.

6
Results

This section discusses the results of the new proposed tools for the HashDAG framework. In particular, the Spheroidal Weathering tool, the Arches simulation tool, and the Combined tool. We cover the visual and performance results for each tool, including the impact of customization options if applicable.

6.1. Visual results

In this subsection, we present the visual results for the Spheroidal Weathering, Arches Simulation, and Combined editing tools. We aim to showcase the visual results of these tools and the impact of their customization parameters.

6.1.1. Spheroidal Weathering

The spheroidal weathering tool can be customized with three parameters; bubble size, bubble shape, and weathering ratio. The details for these parameters are discussed in Section 4.2.

We showcase the impact of all these three parameters. First, we show the impact of bubble size, and then for weathering ratio. For each of these, we use a cube of 40^3. And to illustrate the impact of the bubble shapes, we show the results for the following bubble shapes: cubic, spherical, and flattened spherical bubble shapes. The flattened sphere is implemented by adjusting the weight function of the spherical bubble shape to decrease the weights in the Y-direction.

Bubble sizes In Figures 6.1-6.3 we show the impact of one weathering edit operation with an increasing bubble size, ranging from 7 (radius of 3) to 31 (radius of 15). The weathering ratio has been fixed at 0.5. Each figure makes use of a different bubble shape (cube/sphere/flat sphere).

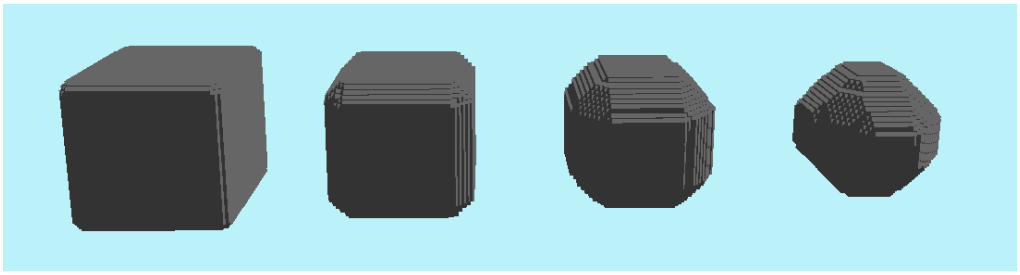


Figure 6.1: Cube-shaped bubble with different bubble sizes, from left to right: 7, 15, 23, 31.

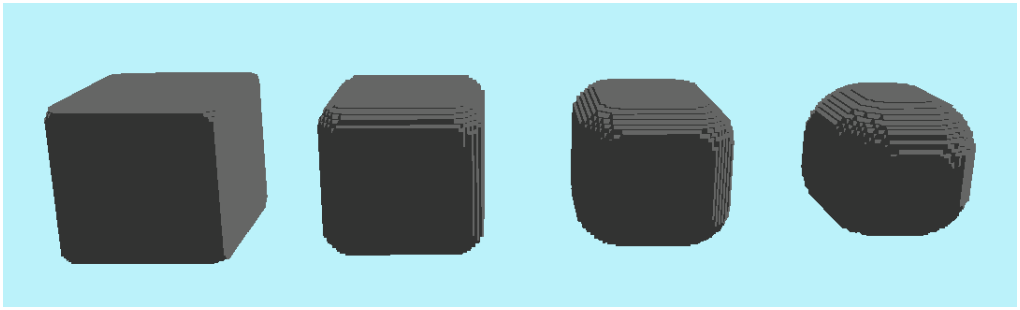


Figure 6.2: Sphere-shaped bubble with different bubble sizes, from left to right: 7, 15, 23, 31.

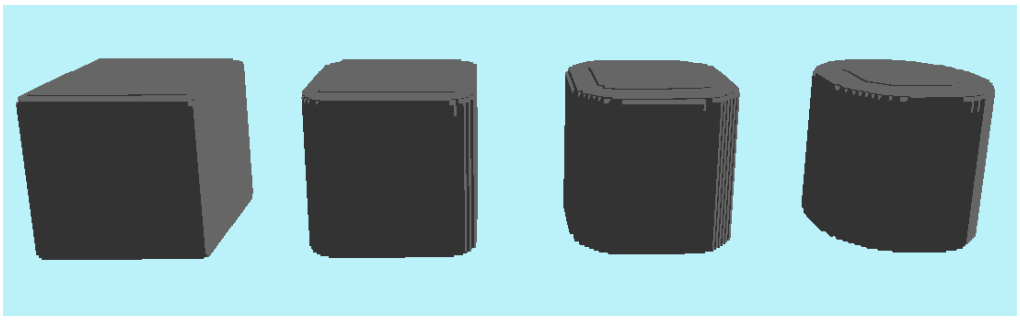


Figure 6.3: Flat sphere-shaped bubble with different bubble sizes, from left to right: 7, 15, 23, 31.

Weathering ratio In figures 6.1-6.3 we show the impact of one weathering edit operation with an increasing weathering ratio, ranging from 0.1 to 0.9. The bubble size has been fixed at 23. Each figure makes use of a different bubble shape (cube/sphere/flat sphere).

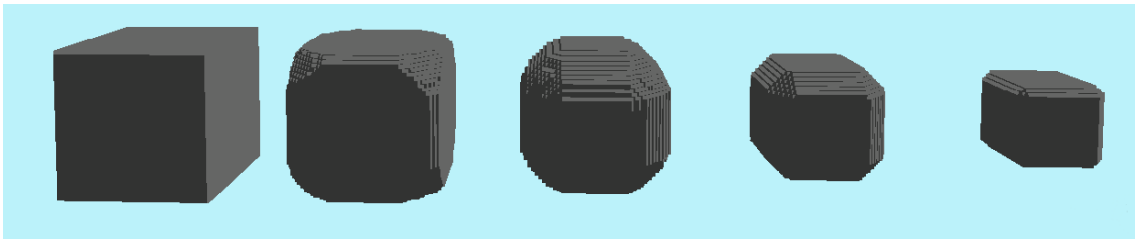


Figure 6.4: Cube-shaped bubble with different weathering ratios, from left to right: 0.1, 0.3, 0.5, 0.7, 0.9

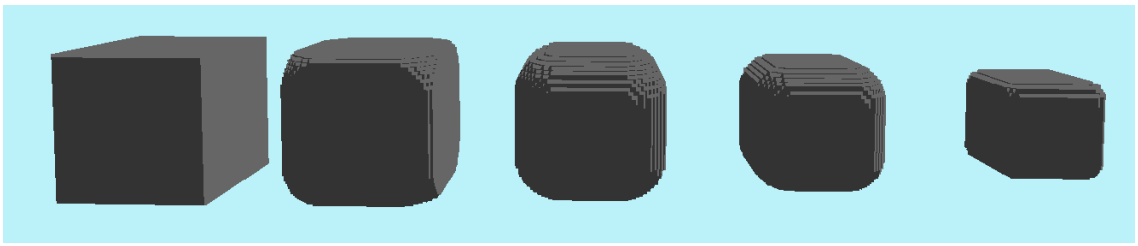


Figure 6.5: Sphere-shaped bubble with different weathering ratios, from left to right: 0.1, 0.3, 0.5, 0.7, 0.9

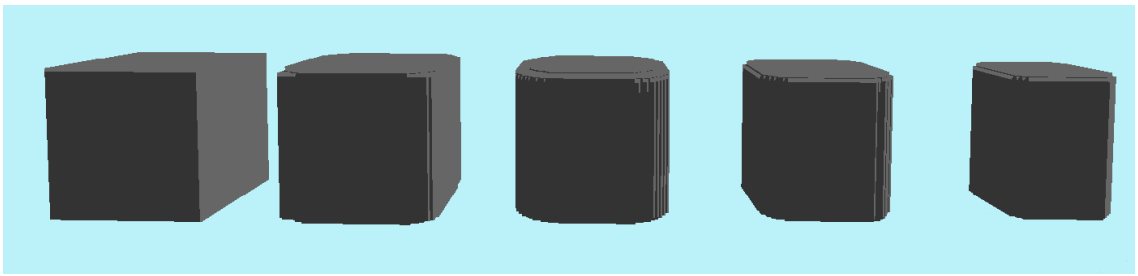


Figure 6.6: Flat sphere-shaped bubble with different weathering ratios, from left to right: 0.1, 0.3, 0.5, 0.7, 0.9

6.1.2. Arches Simulation

The Arches Simulation has a single customization parameter, the repose angle. We have added four different materials with repose angles of 75°, 45°, 35°, and 10°. Figure 6.7 shows the result of performing the Arches simulation tool for these different material types using a cube of 15^3.

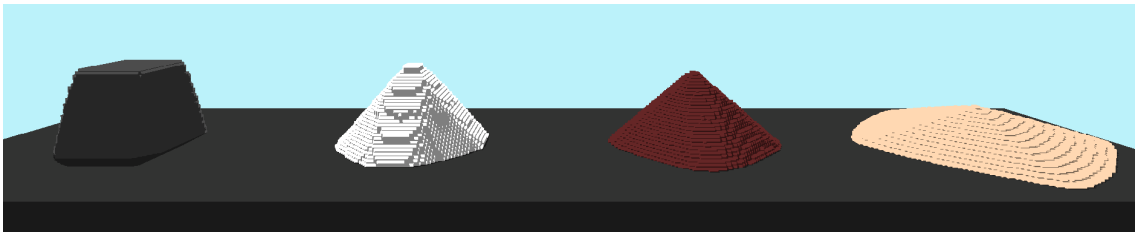


Figure 6.7: Impact of different repose angles on the stabilized granular material.
(From left to right: 75°, 45°, 35°, 10°)

6.1.3. Combined Tool

The combined tool has one customization parameter, debris percentage. It determines what percentage of removed voxels is turned into debris material. This parameter is explained further in detail in Section 4.4. The tool also inherits all customization options from the Spheroidal Weathering and Arches Simulation tools.

We only showcase the debris percentage and weathering threshold parameters (the second being from the Spheroidal Weathering tool) in Figures 6.8-6.11. For all these examples we use a cube-shaped bubble with a size of 31 (radius of 15). In the 4 figures, we show a progressing weathering threshold of 0.3, 0.5, 0.7, and 0.9. For each of these figures, we show the impact of the debris percentage by increasing the debris percentage from 0.1 to 0.9 in steps of 0.2 from left to right.

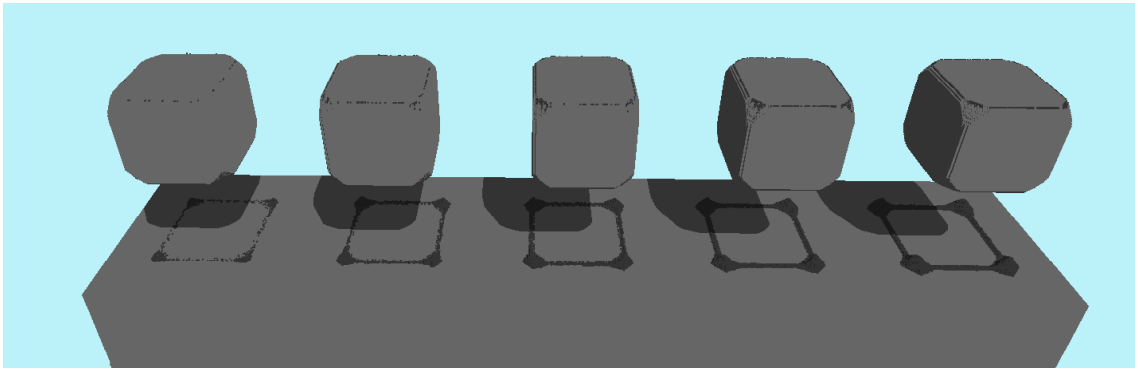


Figure 6.8: Weathering threshold of 0.3. Different debris percentages from left to right: 0.1, 0.3, 0.5, 0.7, 0.9.

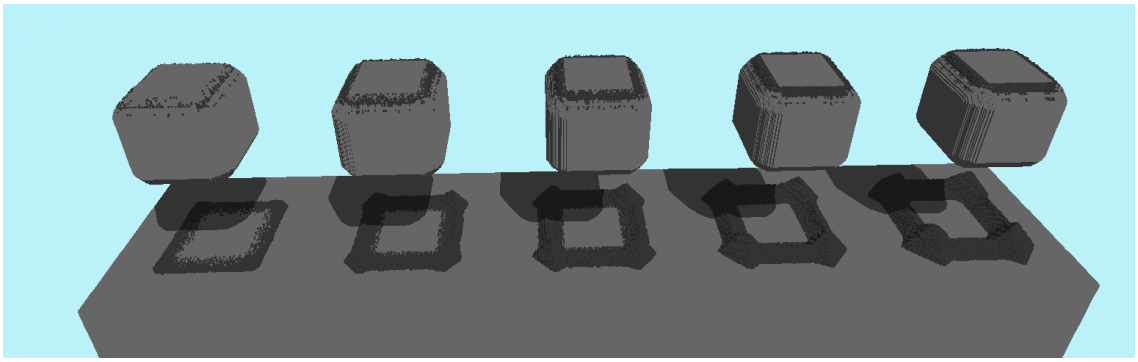


Figure 6.9: Weathering threshold of 0.5. Different debris percentages from left to right: 0.1, 0.3, 0.5, 0.7, 0.9.

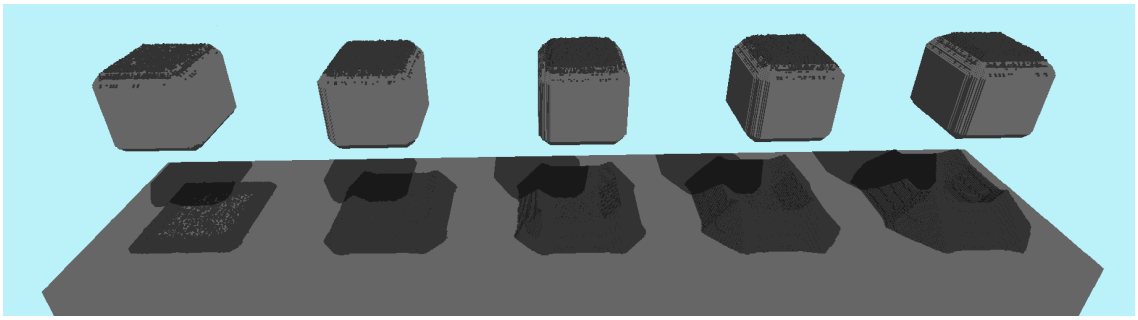


Figure 6.10: Weathering threshold of 0.7. Different debris percentages from left to right: 0.1, 0.3, 0.5, 0.7, 0.9.

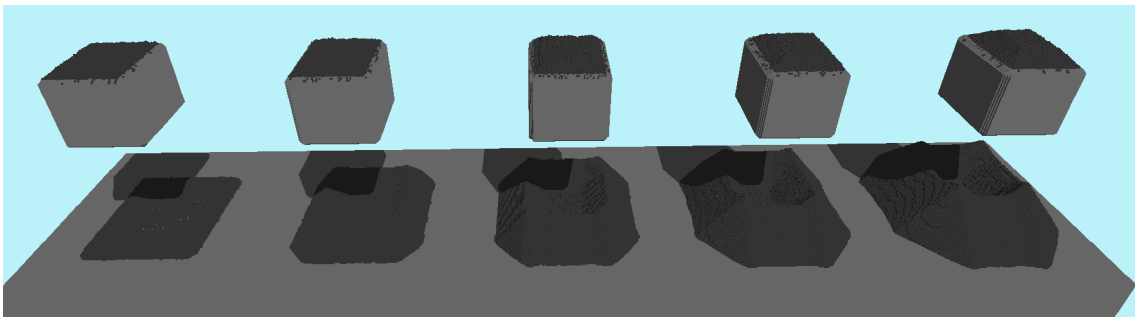


Figure 6.11: Weathering threshold of 0.9. Different debris percentages from left to right: 0.1, 0.3, 0.5, 0.7, 0.9.

6.2. Performance Results

This subsection discusses the benchmarks used to measure the performance of the Spheroidal Weathering and Arches tools. It shows the impact of tool sizes on the run time. Additionally, we aim to present the progression of the run time when using different parameters and to provide insight into the relationship between these parameters and the run time of the tools and corresponding tool steps.

6.2.1. System Architecture

All performance benchmarks were performed on a system using an NVIDIA GeForce GTX 1080 GPU with 8GB of VRAM, an Intel i7-7700K @ 4.20 GHz CPU, and 32GB of 2133 MHz DDR4 memory.

6.2.2. Spheroidal Weathering

For the spheroidal weathering tool, we assess the performance based on the weathering region size (length of one side of the cubic tool region) and the size of the bubble region (length of one side of the bubble region). We use a spherical bubble shape for all experiments, so the bubble is symmetrical.

These experiments are performed on an increasing tool region size, ranging from 41 to 401 with increments of 40 (radius from 20-200 with increments of 20). For the bubble size, we test from 7 to 23 with increments of 4 (radius from 3-11 with increments of 2). The total time measured is the complete time of performing the spheroidal weathering kernel including the time to edit the scene in the HashDAG framework.

The CUDA kernel iterates over all voxels in the tool region and accumulates weighed values to the neighboring values for each solid voxel. Therefore, we expect the results to match the theoretical complexity of $\mathcal{O}(X^3 + S \cdot B)$, where X is the tool size, S is the total number of solid voxels, and B is equal to the total bubble volume ($\texttt{bubbleSize}^3$ when using a symmetrical bubble shape).

Impact of tool size We benchmark the relationship between the runtime and the tool size. Since multiple different bubble sizes were used, the lines are grouped by bubble size. This visualization can be found in Figure 6.12.

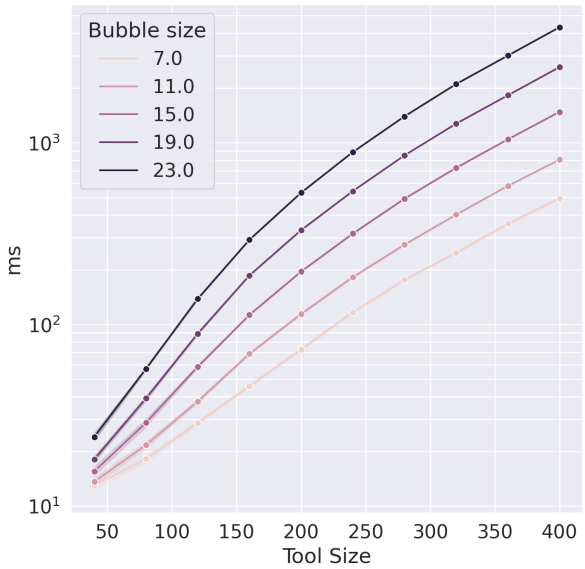


Figure 6.12: Spheroidal Weathering performance using different bubble sizes for progressing tool sizes. Benchmarked in the HashDAG framework.

Using the results in Figure 6.12 we plot with a logarithmic scale. The results confirm the expected theoretical complexity of $\mathcal{O}(X^3 + S \cdot B)$ as we observe a polynomial relationship between the run time and the tool size. We also observe that the bubble size impacts the run time considerably, which is what we look at next.

Impact of bubble size Using the same experiments, we also visualize the relationship between the run time and the bubble size. Similar to the tool size visualization, we have different experiments of the same bubble size but with different tool sizes, so we group by tool size. This visualization can be found in Figure 6.13.

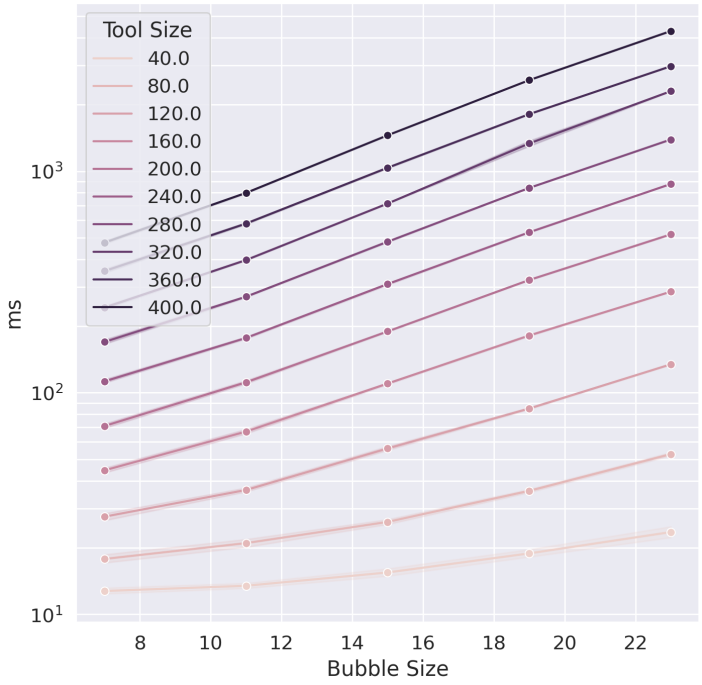


Figure 6.13: Spheroidal Weathering performance using different bubble sizes for progressing tool sizes. Benchmarked in the HashDAG framework.

In Figure 6.13 we plot with a logarithmic scale. This graph also confirms our expectation of the theoretical complexity of $\mathcal{O}(X^3 + S \cdot B)$. We observe a polynomial relationship between the bubble size and the total runtime.

Tool steps breakdown The Spheroidal Weathering part consists of multiple steps. Therefore, we provide a breakdown of the timings of each of these steps. These steps are (1) Initializing the CPU and GPU buffers, (2) running the CUDA Kernel, (3) copying the data from the GPU back to the CPU, and (4) the rest of the time, spent on editing the voxel scene. A larger tool size implies a larger data buffer, (1) leading to more memory to be initialized, (2) more voxels to be processed in the kernel, (3) more data to be copied back from the GPU, and (4) more voxels to be updated. However, we expect only the kernel runtime to be related to the bubble size, as the kernel requires more operations while the buffer size does not change, not influencing the other three steps. We expect the results of the CUDA kernel to match the theoretical complexity of $\mathcal{O}(V + S * B)$ and $\mathcal{O}(V)$ for the other three steps, where V is the total tool volume ($\texttt{toolSize}^3$), S is the total number of solid voxels within the tool region, and B is equal to the total bubble volume ($\frac{4}{3}\pi \cdot \texttt{bubbleSize}^3$ in this case, since we are using a spherical bubble shape).

To plot these four steps, we fix some parameters, therefore we plot the same two plots as before but with fixed parameters. One with a fixed bubble size of 15, and one with a fixed tool size of 240.

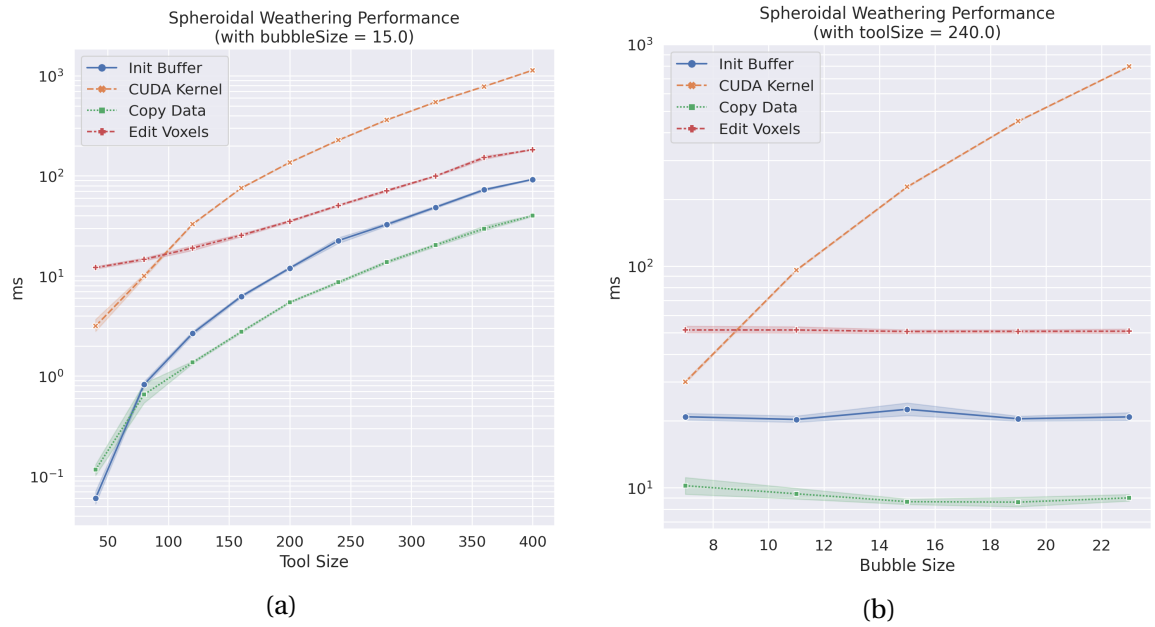


Figure 6.14: Runtime of Spheroidal Weathering steps. Figure (a) increases the tool size and fixes the bubble size on 15. Figure (b) increases bubble size and fixes tool size on 241.

In Figure 6.14(a) and (b) we visualize the run time of the different Spheroidal Weathering steps for increasing tool- and bubble size.

In Figure 6.14(a) we observe the change of run time of all steps with a larger tool size. As we plot with a logarithmic scale, we see a polynomial relationship between the run time and tool size. We can conclude that all four steps of the weathering tool are related to the size of the tool region and slow down with a larger tool region. We can also conclude that the CUDA Kernel has by far the largest runtime of all steps.

In Figure 6.14(b) we observe the change of run time of all steps with a larger bubble size. We see a polynomial relationship between the CUDA Kernel step run time and the tool size. This is due to the number of voxels to process in the kernel being much higher with larger bubbles. We see that all other steps do not change depending on the run time of the bubble size, meaning that these are only influenced by the size of the tool size and not bubble size.

Conclusion For the Spheroidal Weathering tool, we see that the CUDA kernel takes the most time by far (Figure 6.14). This is expected as most operations are performed in the CUDA kernel. The other steps, editing the voxels, and initializing and copying data from and to the GPU take less time. We also noted that increasing the bubble volume has a significant effect on the run time of the weathering tool. This is due to the tool having to add much more values in the buffer with increased buffer size. In Figure 6.14, we see that this increase in runtime is only caused by the kernel taking longer, the other three steps are not influenced by a larger bubble volume.

6.2.3. Arches Simulation

The Arches Simulation benchmark performance is assessed using offline simulation benchmarks and fully integrated benchmarks inside of the HashDAG application.

Simulation scenarios We briefly discuss the best and worst scenarios for the Arches simulation and the benchmark scenario we chose. Figure 6.15 illustrates these different scenarios. Figure 6.15(a) shows a worst-case scenario where material slides down a slope indefinitely. Figure 6.15(b) shows a case where all material falls towards the center and stabilizes quickly. Figure 6.15(c) shows the best-case scenario, in which no simulation is even performed, the material only falls down and stabilizes immediately. All these cases have greatly varying results when benchmarked, therefore we decided on using a simple scenario for benchmarks. Figure 6.15(d) shows the scenario we use for benchmarks, which allows the material to freely fall down and to the sides. While this scenario does simplify real use cases, which can result in much more varying runtimes depending on the terrain, these benchmarks do give an indication of what to expect from the runtime when adjusting the tool size.

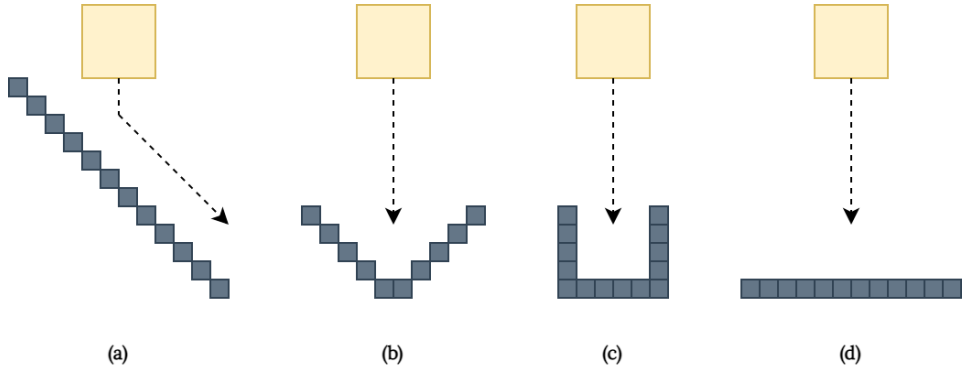


Figure 6.15: Illustration of the worst (a), good (b), best (c) scenarios, and the scenario we use for benchmarks (d).

Offline and In-Application benchmarks To test the performance of the Arches simulation itself, we perform the simulation outside of the HashDAG application. These "offline" benchmarks aim to show the impact of the amount of granular material on the simulation runtime without any impact from the HashDAG application. For these offline benchmarks, we prepare all material layers and only run the granular simulation. We compare this offline benchmark with the in-application benchmark which - in addition to the simulation - also initializes the granular material layers based on the voxel scene, generates new layers during simulation, and updates the voxel scene.

Beneš and Forsbach [2] made use of a similar simulation method that noted a $\mathcal{O}(k + X^2)$ complexity where k is the number of layers, and X is the size along a single axis. However, this method only seemed to note the complexity of a single iteration with a fixed number of layers across the whole simulation region. The runtime complexity until stabilization is more involved, however, since as material spreads, more simulation steps are needed. We hypothesize the total runtime until stabilization to linearly increase with the volume of the granular material that is added to the scene. And thus polynomially related to the tool size, $\mathcal{O}(X^3)$ with X being the tool size.

We generate cubes of granular material on top of a single solid layer, with cubes of X^3 where the cube size X ranges from 5 to 50 with intervals of 5. We perform the simulation with a repose angle of 45 degrees until the simulation stabilizes using a deposit threshold of 0.1, and repeat for 50 runs. The results of these experiments can be found in Figure 6.16.

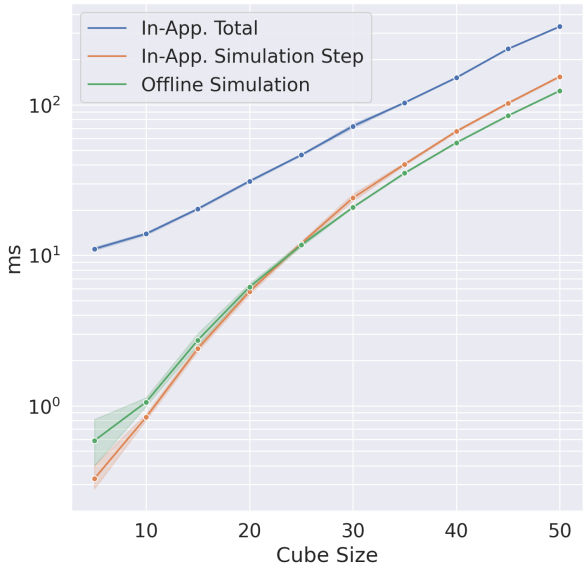


Figure 6.16: Offline and in-application runtime of the Arches simulation method.
50 simulations per datapoint, 45-degree repose angle, simulated until stabilized with a deposit threshold of 0.1.

Figure 6.16 shows the relation between the tool size and the runtime of the offline, the simulation step of the in-application tool, and for reference, the total time of the in-application tool. The graph has a logarithmic scale, as expected, we observe a polynomial relationship between the tool size and the runtime of the Arches simulation method.

We see that the total runtime of the in-application tool is higher than both the in-application and offline simulation runtime. This is because the complete in-application tool also initializes and generates stacks based on voxel data, as well as updates the voxel scene with the stabilized material. We look at these individual steps next.

Benchmarks of In-Application Steps In order to get the runtime of the complete Arches simulation tool in use in the application, the Arches simulation tool is benchmarked in the HashDAG framework in a reproducible setting with each step being timed separately.

To test the Arches simulation tool in the application, we first generate a flat surface using a large cube somewhere in an empty space of the scene. Above this surface, a cube of granular material is generated. The benchmarks are set up similarly to offline benchmarks: we generate increasingly larger cubes of X^3 where the cube size X ranges from 5 to 50 with intervals of 5. For each, we perform the simulation with a repose angle of 45 degrees until the cube stabilizes using a deposit threshold of 0.1, and repeat for 50 runs.

As this benchmark is performed in the HashDAG application, three more stages are performed in addition to the Arches simulation: (1) Generate the Arches data structure and add the granular material based on voxel data, (2) generating stacks during simulation in uninitialized areas, and (3) convert the results into voxels and update the HashDAG structure accordingly. The results of these benchmarks are shown in Figure 6.17.

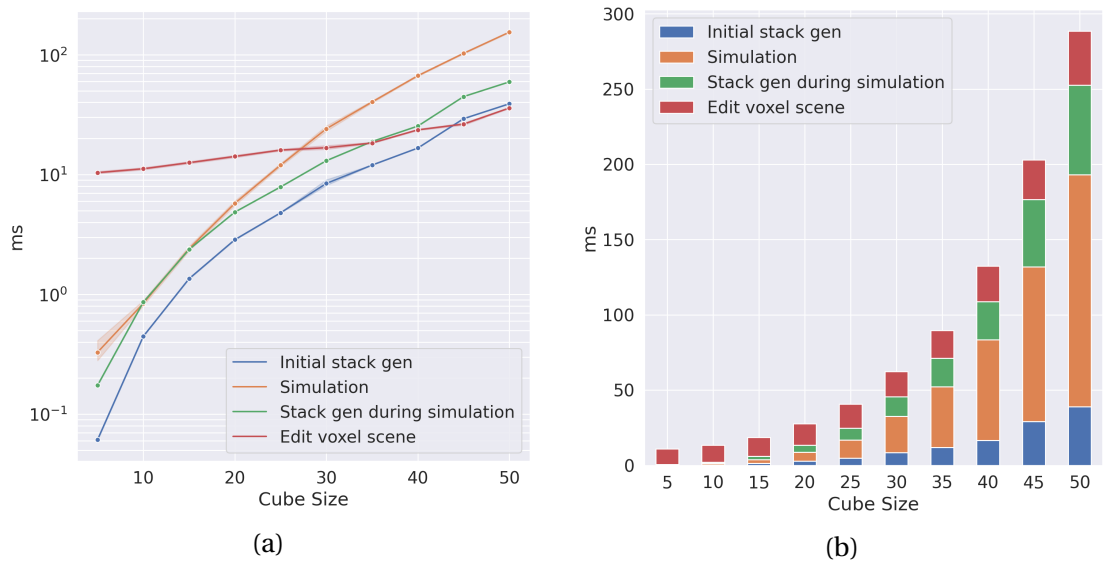
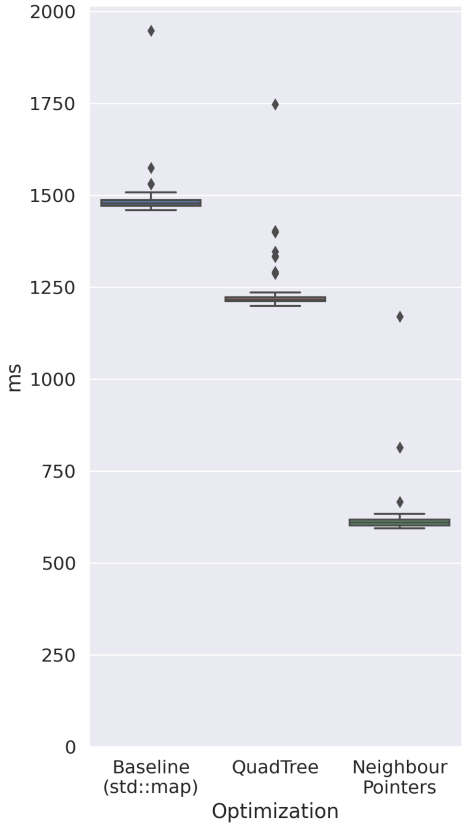


Figure 6.17: Runtime of the different Arches Simulation stages, benchmarked in the HashDAG framework.

From Figure 6.17 we can conclude that the simulation step has the slowest run time. And just like with the offline benchmarks, the simulation run time has a polynomial relationship with the cube size. Adding material and editing the voxel scene stages also show a polynomial relationship with the cube size, although being much faster than the simulation stage. The edit scene stage seems to have a higher runtime with smaller cube sizes but seems to increase slower than the other stages, while also being polynomial in terms of cube size.

QuadTree Optimizations After the development of the Arches simulation tool, we still found more room for performance improvements. Apart from some general C++ optimizations to reduce copy operations, for example, we had two major optimizations. First, the use of a QuadTree instead of using a `std::map` data structure for the grid data. And secondly, reducing the number of grid fetches by storing, for each grid cell, pointers to its neighbors.

To display the improvements, we performed a large benchmark experiment with a setup similar to the offline benchmarks. For this single large benchmark, we make use of a cube size of $X = 50$ with a repose angle of 45 degrees, simulated until stabilized with a deposit threshold of 0.1. All three benchmarks have been run 50 times.



Optimization	Median (ms)
Baseline (`std::map`)	1477.8
QuadTree	1214.9
Neighbor Pointers	610.3

Figure 6.18: Impact of QuadTree and neighbor pointer optimizations. These comparisons were run on a prior version without other improvements. Therefore, the runtime of these comparisons in ms does not correspond to the other benchmarks. This is purely to show the relative improvements.

Figure 6.18 shows the improvement from both the QuadTree and neighbor pointer optimizations. The change from a `std::map` resulted in a 1.21× speedup (based on the median runtime of `std::map` compared to QuadTree). The neighbor pointers optimization resulted in a 1.99× speedup compared to the QuadTree (based on the median runtime of QuadTree compared to neighbor pointers optimization). The total speedup of both optimizations compared to using `std::map` is 2.42×.

Optimization: Active Stacks For the proposed Arches simulation tool, we make use of a bounding box to specify a region of granular material to simulate. In practice, not every single stack inside the bounding box contains granular material. On top of that, stacks inside of the granular material bounds could already be stabilized, while stacks around the outsides are not. Therefore, we implemented the active stacks system as described in Section 4.3.

Conclusion The first thing that we notice from the Arches simulation run times is that the run time of the tool is quite a bit slower than the Spheroidal Weathering tool for similar tool sizes. We should also keep this in mind for the combined tool performance, where the Arches simulation is the main bottleneck compared to the Spheroidal Weathering. We also performed a breakdown of the different steps of the Arches Simulation tool. We see that as expected, the total run time and all individual steps increase polynomially in relationship to the tool size. For the individual steps (Figure 6.17), we see that the simulation step takes the longest. The other three steps - initializing material stacks, generating stacks during simulation, and editing voxels in the scene - took a relatively shorter time.

6.2.4. Combined Tool

The combined tool is a combination of the Spheroidal Weathering and Arches tools which both have their own parameters. So in order to analyze the performance of the combined tool, we fix as many parameters in order to isolate the parameters related to the combined tool. Therefore we use a fixed cube size of 50 to match the largest Arches simulation benchmark case as opposed to an increasing cube size. For the fixed parameters for Spheroidal Weathering and Arches, we use parameters in the middle of the range of parameters that were used for the individual tool benchmarks, resulting in the following fixed parameters:

- **Global**:
  - Cube size: 50
- **Spheroidal Weathering**:
  - Bubble size: 11
  - Weathering threshold: 50
- **Arches Simulation**:
  - Repose angle: 45 degrees
  - Deposit threshold: 0.1

This leaves only one parameter for the combined tool, debris percentage. For this benchmark, we progress through debris percentages of 0.1 to 0.9 in steps of 0.2 to cover the whole range of values. For each case, we perform 5 benchmarks and plot the average runtime with a 95% confidence interval of the steps of the Spheroidal Weathering tool, the steps of the Arches Simulation tool, and the total time of the Spheroidal Weathering and Arches tools. These results can be found in Figure 6.19.

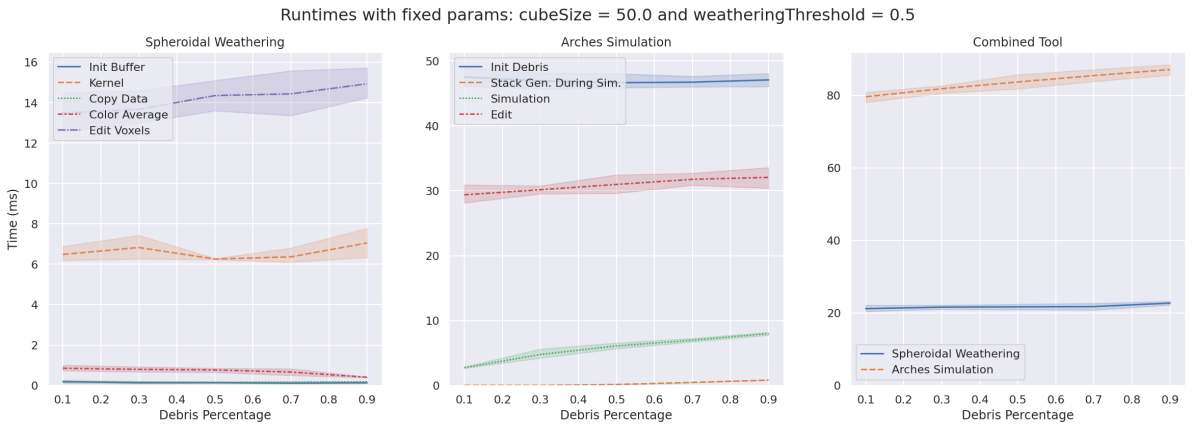


Figure 6.19: Runtime of Spheroidal Weathering and Arches tool steps in the combined tool. Fixed cube size and weathering threshold, with increasing debris percentage.

Figure 6.19 shows that the debris percentage has very little impact on the runtime of the Spheroidal Weathering tool. This is because the debris percentage parameter impacts the amount of debris to simulate, which does not impact the number of operations for the Spheroidal Weathering tool.

For the Arches simulation tool, we observe a slight increase in the simulation time and the time to generate stacks during simulation when increasing the debris percentage. The edit and initial stack generation steps are relatively flat, the increased amount of debris does not seem to affect

these steps much. With a higher debris percentage, more debris material is initialized and simulated. We hypothesize that with an increased weathering threshold, more debris material is created, resulting in a higher disparity between weathering and Arches. Therefore we fix the debris percentage at 50% and increase weathering threshold from 0.1 to 0.9 with steps of 0.2 to see the effects of more debris material:

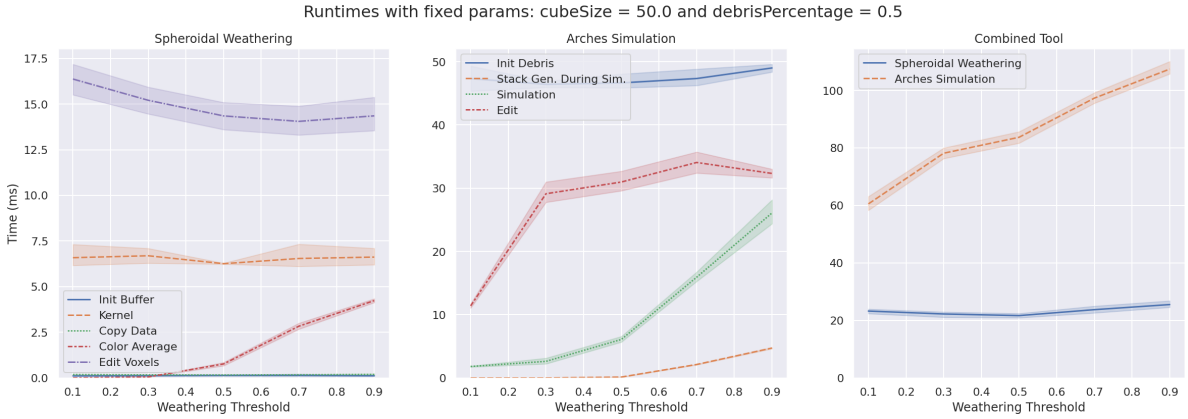
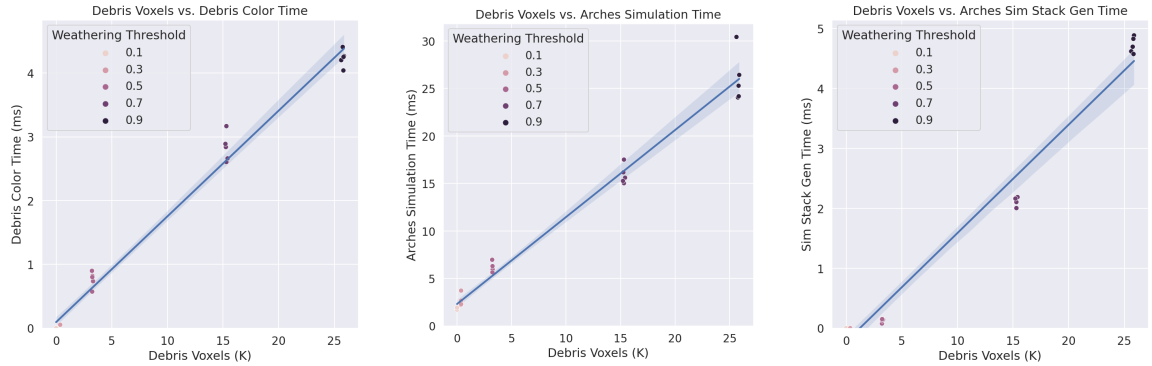


Figure 6.20: Runtime of Spheroidal Weathering and Arches tool steps for fixed cube radius and debris percentage, with increasing weathering threshold.

Figure 6.20 shows the same metric, for an increasing weathering threshold. We see that the time it takes to get the colors for the debris voxels takes longer with more debris material. We also see a very clear increase in the Arches simulation time and the time to generate stacks during simulation. To confirm, we plot the number of debris voxels against the Arches simulation time to confirm this relation. We fix the debris percentage to 50% and plot the Arches simulation time against the number of debris voxels.



(a) (b) (c)

Figure 6.21: Runtime of debris coloring, the Arches simulation, and Arches stack generation during simulation in relation to the number of debris voxels (in thousands).

Figure 6.21 plots the number of debris voxels against the run time for averaging debris color, the Arches simulation, and generating stacks during the Arches simulation. As the generation of debris voxels has a random element to them, we use a scatter plot for the data points and plot a regression line over these data points. We see that there is a linear relationship between debris voxels for each of these steps. Meaning that all of these steps are dependent on the number of debris voxels, and thus increase along with the parameters that also increase the amount of debris, like debris percentage and weathering threshold.

Conclusion The Combined Tool depends on more than just the tool size parameter as an input. This tool combines both the Arches Simulation and Spheroidal Weathering, and therefore also inherits the parameters from those tools. We already know the performance impact of the tool size on the Arches Simulation and Spheroidal Weathering tools. As we already benchmarked these two, we fixed most parameters related to these tools and only focused on the debris percentage parameter from the Combined Tool. As the Combined Tool consists of a combination of the Spheroidal Weathering and the Arches Simulation tool, when talking about the performance of the Combined Tool, we used some of the same metrics as with the Spheroidal Weathering and Arches Simulation benchmarks.

We benchmarked the Combined Tool by increasing the debris percentage. This had little impact on the Spheroidal Weathering run time but did result in slower run times for the Arches Simulation. We hypothesized this was because of the number of total debris voxels created by weathering. Therefore we also benchmarked with a fixed debris percentage while increasing the weathering threshold to create more debris. We saw a clear increase in run time for the Arches simulation, mainly simulation and generating stacks because more stacks need to be generated and simulated. Additionally, we noted that the debris color averaging method takes longer because more voxel colors have to be fetched. We saw a clear linear relationship between the number of debris voxels and the run time of these steps. We can conclude that the performance of the Combined Tool linearly increases with tool size and the number of debris voxels. Since “debris voxels” is not a parameter but dependent on other parameters, all parameters that lead to an increased amount of debris material also lead to an increase in the runtime of the Combined Tool.

7

Discussion

This section covers some discussion topics about the Spheroidal Weathering and Combined Tool.

7.1. Spheroidal Weathering

Traversal over solid voxels Currently, the tool region iterates over each voxel, including empty voxels which are irrelevant to accumulate and weighing neighbors. It then checks the value at that voxel, and skips if it is empty. This can be improved by traversing the HashDAG nodes and leaves instead of each voxel, thus only processing the solid voxels. However, as this requires significant changes to the CUDA kernel, we decided not to implement this and leave this for future work.

CUDA Thread Synchronization To continue on the previous paragraph, some of the threads in the CUDA kernel of the Spheroidal Weathering tool are terminated earlier than others because threads that process empty voxels are terminated early. This leads to some threads finishing before others. Additionally, some threads can finish sooner because part of the neighboring voxels in the bubble region is empty. The kernel can then skip calculating the weight values for these voxels and finish quicker. The differences in synchronization can lead to performance fluctuations depending on how many threads terminate quicker, however, we deem that details about thread synchronization are out of scope for our thesis and leave more in-depth analysis for future work.

7.2. Combined Tool

Volume preservation for debris material The Arches simulation converts material stacks into voxels. When using the combined tool, weathered voxels are transformed into material stacks. When the simulation finishes, these material stacks are discretized back into voxels, based on the material type that intersects the center of each voxel, as discussed in Section 4.3.6.

In certain cases, material stacks do not fill a voxel halfway, resulting in an empty voxel after discretization. Subsequently, the volume of the debris voxels after the simulation might be smaller than the volume before the simulation. This can occur when using a low repose angle, for example, resulting in material stacks not crossing halfway.

However, we view this as a minor drawback, given the benefits of fractional values for material heights. Solving this problem to retain all debris material would require the simulation of each individual voxel.

8

Limitations

This section discusses the limitations for each of the three proposed editing tools: Spheroidal Weathering (Section 8.1), Arches simulation (Section 8.2), and the Combined Tool (Section 8.3).

8.1. Spheroidal Weathering

Memory storage method limits tool size The Spheroidal Weathering CUDA kernel uses a 3D buffer with a value for each voxel in the editing region, which can be inefficient for larger areas with relatively large empty spaces. As a result, the total size of the buffer is a limiting factor for the total tool size. This can be challenging to address without implementing a novel method or using a hierarchical data structure within the CUDA kernel. For larger empty spaces, using a compressed hierarchical data structure is more efficient. We have left this optimization for future work.

8.2. Arches Simulation

Parallelization The Arches simulation runs on a single thread on the CPU, limiting its performance at a larger scale. The Arches simulation can be run in parallel, with some difficulties, however. Simulating material is dependent on neighboring stacks to proportionally transfer material, as well as potential material falling on top of the stack. This is no problem when running on a single thread as each stack is simulated after another.

To run the Arches simulation in parallel, you need to ensure that neighboring stacks are not run in parallel. A valid solution is to simulate chunks of stacks in such a way that the material that falls outside of the chunk boundaries does not fall into another chunk.

Material stacks are voxels in subsequent edits Once the material is stabilized, it is converted into voxels, further simulation operations do not take the granular material from previous Arches simulations into account and the voxels are observed as solid. This can cause unexpected behavior if the material is expected to be used in the simulation again. This is because of the way edit operations are handled in the HashDAG application, the tool finishes and updates the voxels before starting a new one. A potential naive option is to persist all the material stacks, which is possible with the current data structure. However, the HashDAG application includes more than editing tools that edit other voxels, possibly containing voxels of the material stacks. This means that each editing tool also needs to update the persistent data structure of the Arches material stacks. However, in an application without any other editing tools, this is possible.

Material stack generation over overhangs To optimize material stack generation, the Arches simulation tool generation material stacks from voxels inside the tool regions and the regions below

until a solid voxel is hit. This is to avoid unnecessary operations of reading all voxels to the bottom of the scene. This approach works fine for flat surfaces but can lead to problems when overhangs are present below the tool region and solid voxels.

Figure 8.1 illustrates an example where base height layers are generated in place of empty voxels. In this case, granular material is able to fall into the space below overhangs. A solution is to regenerate base height stacks using voxel data whenever a layer can transfer material into a base height layer. However, this further complicates the stack generation method during the simulation and therefore was not implemented. If such a situation arises, the material does not fall into or below the overhang.

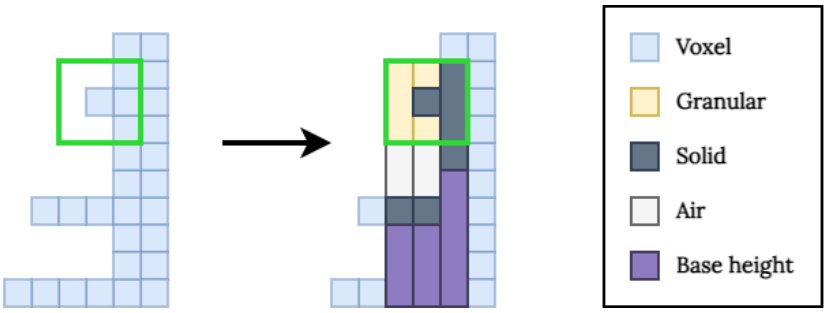


Figure 8.1: Limitation of stack generation method, generating base height layers at overhangs where air voxels are present.

Stabilized state relies on the order of simulation A common technique for simulations is using a ping-pong method by having two data structures, one to read from and one to write to, which alternate. However, the Arches simulation moves material based on the repose angle of material layers. Using a ping-pong approach naively means that multiple stacks continue depositing material to the same stack in the second grid, the material may already be moved past the repose angle. Therefore, we decided to implement the simulation such that it reads and writes to the same grid. This simplifies the simulation algorithm as we can simply iterate over all active stacks and perform simulation on these stacks while ensuring that the repose angle is held. The main implication is that a different order of simulation can result in a different simulation result, whereas this usually isn't the case with simulation methods. The prior works utilizing this method did not mention anything regarding this topic. It is a minor limitation but something to keep in mind when using the simulation tool.

8.3. Combined Tool

Debris color A limitation of the combined tool is the inaccuracy and performance impact of determining the debris colors. Due to the nature of the material stacks of the Arches simulation method, it is not possible to retain the color of individual voxels. Voxels on top of one another are merged into material stacks, and these voxels may all have different colors.

A basic solution is implemented using the average color of all debris voxels. A large region of color from the voxels is averaged, which leads to a loss of detail when applied on a large scale. We fetch the colors of all debris voxels and average all, leading to some performance drawbacks with a large number of debris voxels. However, we do not consider this a major issue as this is simply a rudimentary solution to get somewhat accurate colors. We also indicate the impact of this method of averaging debris colors in the benchmark results so that it is clear what speedup can be achieved if improved.

9

Conclusion

Various works have explored the topic of high-resolution voxel scenes. The HashDAG framework allows for real-time editing of these high-resolution voxel scenes. These editing tools are impressive, but we can expand upon them. Researchers have published various works that use voxel-based editing methods. This work focuses on adding more complex, simulation-based editing tools. We have integrated three different editing tools into the HashDAG framework.

The first contribution is the Spheroidal Weathering tool where we used the concepts of the terrain weathering methods from Beardall et al. [1] to create a customizable terrain weathering tool. We discussed the limitations, assumptions, and required changes in order to integrate this method into the HashDAG application. We have had to simplify the method to change it from an iterative method to one that provides a weathered result with a single click. This tool is implemented on the GPU using CUDA. Increasing the tool and bubble size polynomially increases the runtime of the Spheroidal Weathering tool. We have shown that a Spheroidal Weathering tool can edit terrain interactively and close to real time for smaller tool sizes. This tool allows various customization options (like weathering threshold) and bubble properties (like shape and radius) to get different weathering behavior.

The second contribution is the Arches simulation tool, where we use the layered granular simulation method of the Arches framework [21]. We generate material in a region of the scene from the HashDAG voxel data, inside the tool region. The simulation region expands when material falls outside of initialized stacks during simulation. We provide a choice of various different material types with different angles of repose. To avoid unnecessary simulation iterations, we only simulate active stacks, referring to stacks that are not yet stabilized. We have shown that this simulation method can work well and can interactively add and simulate granular material interactively to the HashDAG scene, however, does slow down considerably on a larger scale. The method still has some potential with regard to performance in terms of stack generation and parallelization.

The third and final contribution is the Combined tool, combining both the Spheroidal Weathering and the Arches simulation tools to perform terrain weathering with debris simulation. The tool first performs weathering, after which it generates material layers based on the removed voxels to simulate the debris material. This implements the missing debris simulation feature of the spheroidal weathering work. This tool enables the same customization options as the two other tools with additional control over the debris amount, leading to fine-grained control over the weathering and debris simulation. We have shown that the Spheroidal Weathering and Arches simulation can be combined into a novel editing tool to interactively weather terrain and simulate debris.

To summarize, we explored various potential methods for creating editing tools for the HashDAG framework. Then we identified two methods to serve as a base for new editing tools. We adapted these methods to enable integration into the HashDAG framework. We presented the issues with

these methods, including the required changes to solve these. Then, we presented a novel editing tool that combined these two methods. We implemented and integrated three editing tools in the HashDAG framework. This results in three editing tools that edit the voxel scene interactively, and close to real time depending on the tool size. These tools still have more room for improvement in terms of performance, with plenty of ideas presented in future work.

10

Future Work

This chapter discusses future work, largely based on the limitations discussed in Chapter 8. We discuss future work for the three proposed editing tools: Spheroidal Weathering (Section 10.1), Arches simulation (Section 10.2), and the Combined Tool (Section 10.3).

10.1. Spheroidal Weathering

Hierarchical data structure One of the limitations of the Spheroidal Weathering tool is how it stores data about the number of neighboring voxels, using a 3D buffer without compression. Future work may involve implementing a hierarchical data structure, such as an octree, or using the HashDAG data structure as used in the voxel scene.

DAG traversal for accumulation and weighing neighbors Another potential direction for future work is to traverse the hierarchical DAG data structure instead of iterating through all voxels in the tool region. This decreases the number of operations for regions with large empty spaces, improving performance.

Pre-computing neighbor weights Another potential optimization is pre-computing the weighed neighbors for each voxel so we can look up the accumulation of weighed neighbors for each voxel while editing the voxel scene. This requires limiting customization options, as different customization parameters influence the weight of neighbors. This pre-computation improves the performance of the edit operations but also increases memory usage as this data needs to be stored as well.

Separable kernels The CUDA kernel of the Spheroidal Weathering method is a convolution kernel over all voxels in the bubble. An optimization method for convolution kernels is the use of separable kernels which decomposes kernels into several one-dimensional kernels. This would be an interesting field to explore for increased performance.

Using compressed 64-bit HashDAG leaves A different direction for future work is to make use of the compressed 64-bit leaves of the HashDAG. These leaves compress a 4x4x4 area into 64-bit values. By fetching a leaf for this 4x4x4 area, the number of voxels inside bubbles can be calculated with fewer voxel fetches. Using clever bitwise operations, you could reduce the number of voxel fetches. This method does become harder with arbitrary weight functions that can assign any weight to any voxel, complicating the bitwise operations drastically. Therefore, we decided not to delve further

into these options but, with simplified weight functions, this is a viable option. Especially, simplifying to a cubic bubble lends well to this optimization.

10.2. Arches Simulation

Parallelization The most notable limitation of the current Arches Simulation is its performance at scale, as the simulation runs on a single thread on the CPU. A suggestion for future work is to divide the simulation into separate chunks that can be parallelized. The main difficulty is the fact that multiple stacks can deposit material onto the same stack. Additionally, it is important to keep in mind that stacks are generated during simulation. If the simulation is adjusted such that chunks of stacks are simulated in parallel and do not deposit to or generate the same stack, it should be possible to parallelize the simulation method.

Adaptive stack generation beneath overhangs A limitation of the current Arches simulation implementation is that stacks below overhangs are not generated if they are already below the tool region. Figure 8.1 shows an example. For future work, the stack generation method during simulation can be improved so that material layers are generated from the voxels in these cases. This allows material to fall into overhangs anywhere in the simulation.

Stack generation using HashDAG leaves The HashDAG data structure compresses 4^3 voxel regions as 64-bit values. These can be used to improve the performance of the stack generation method by creating chunks of 4×4 stacks at a time, without the need to fetch each voxel separately.

Material stacks to voxel conversion The Arches granular simulation editing tool converts the granular layers to voxels after simulating the layers. As these are converted to voxels in the HashDAG scene, they are considered solid voxels and are not considered granular material for subsequent editing operations.

A recommendation for future work is to maintain the data structure that is currently used across multiple simulations, and spanning the entire scene. Another potential solution is to store the granular material type for each voxel as an attribute in the HashDAG data structure, similar to how color data is stored. For subsequent Arches editing operations, generating material stacks can use this new attribute to generate existing granular material.

This method needs to work together with the other HashDAG editing tools; either by updating the Arches material stacks or attributes after other edit operations or by disabling other edit operations.

Adaptive deposit threshold The deposit threshold is used to determine when to stop depositing material from stacks. We note that using the same deposit threshold can result in different repose angles when the material stabilizes. A potential solution is to use an adaptive deposit threshold, proportional to the repose angle. The findings of the 45-degree angle case - with a repose height of 1.0 - suggest using a deposit threshold of 10%, given the same tradeoff between performance and accuracy. We have experimented with this adaptive method, which gave reasonably good results. In the end, we decided on a fixed deposit ratio of 0.1 for all repose angles in order to simplify the method and further benchmarks. However, we note that for future work, this is a fairly simple change to make and does not influence the proposed method substantially.

10.3. Combined Tool

Debris color Ideally, the color of each individual voxel should transfer to the debris voxels. This problem can be solved by either integrating color data for individual voxels into the Arches simulation method or by using a different method.

We do have one idea to maintain the original colors of the debris voxels without blending colors for the whole region and without adjusting the Arches simulation method. By storing the colors of each debris voxel from the original voxel scene. Once the debris simulation is completed, randomly assign a voxel color to the debris from the list of original voxel colors. This approach preserves the original colors of the scene, but without the guarantee that the color matches up to the position where the initial debris voxel falls to. As this is no perfect solution, this is still an interesting topic for further research.

Bibliography

[1] M. Beardall, M. Farley, D. Ouderkirk, C. Reimschussel, J. Smith, M. Jones, and P. Egbert. Goblins by Spheroidal Weathering. page 8, 2007. URL https://dl.acm.org/doi/10.5555/2381384.2381386.

[2] Bedřich Beneš and Rafael Forsbach. Layered data representation for visual simulation of terrain erosion. In *Proceedings Spring Conference on Computer Graphics*, pages 80–86, Budmerice, Slovakia, 2001. IEEE Comput. Soc. ISBN 978-0-7695-1215-0. doi: 10.1109/SCCG.2001.945341. URL http://ieeexplore.ieee.org/document/945341/.

[3] Bedřich Beneš and Rafael Forsbach. Visual Simulation of Hydraulic Erosion. page 8, 2002.

[4] Bedřich Beneš and Toney Roa. Simulating Desert Scenery. page 6, 2004.

[5] V. Careil, M. Billeter, and E. Eisemann. Interactively Modifying Compressed Sparse Voxel Representations. *Computer Graphics Forum*, 39(2):111–119, May 2020. ISSN 0167-7055, 1467-8659. doi: 10.1111/cgf.13916. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13916.

[6] Yanyun Chen, Lin Xia, Tien-Tsin Wong, Xin Tong, Baining Guo, and Heung-Yeung Shum. Visual Simulation of Weathering By γ-ton Tracing. page 7, July 2005.

[7] Nuttapong Chentanez and Matthias Müller. Real-Time Eulerian Water Simulation Using a Restricted Tall Cell Grid. page 10, July 2011.

[8] Bas Dado, Timothy R. Kol, Pablo Bauszat, Jean-Marc Thiery, and Elmar Eisemann. Geometry and Attribute Compression for Voxel Scenes. *Computer Graphics Forum*, 35(2):397–407, May 2016. ISSN 01677055. doi: 10.1111/cgf.12841. URL http://doi.wiley.com/10.1111/cgf.12841.

[9] McKay T. Farley. Fast Spheroidal Weathering with Colluvium Deposition. page 43, November 2011.

[10] M.D. Jones, M. Farley, J. Butler, and M. Beardall. Directable Weathering of Concave Rock Using Curvature Estimation. *IEEE Transactions on Visualization and Computer Graphics*, 16(1):81–94, January 2010. ISSN 1077-2626. doi: 10.1109/TVCG.2009.39. URL http://ieeexplore.ieee.org/document/4815233/.

[11] Balázs Jákó. Fast Hydraulic and Thermal Erosion on GPU. page 8, 2011.

[12] P. Krištof, B. Beneš, J. Křivánek, and O. Šťava. Hydraulic Erosion Using Smoothed Particle Hydrodynamics. *Computer Graphics Forum*, 28(2):219–228, April 2009. ISSN 01677055, 14678659. doi: 10.1111/j.1467-8659.2009.01361.x. URL https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2009.01361.x.

[13] V. Krs, T. Hädrich, D. L. Michels, O. Deussen, S. Pirk, and B. Benes. Wind Erosion: Shape Modifications by Interactive Particle-based Erosion and Deposition. *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation - Posters*, page 3 pages, 2020. ISSN 1727-5288.

doi: 10.2312/SCA.20201216. URL https://diglib.eg.org/handle/10.2312/sca20201216. Artwork Size: 3 pages ISBN: 9783038681199 Publisher: The Eurographics Association Version Number: 009-011.

[14] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. High resolution sparse voxel DAGs. *ACM Transactions on Graphics*, 32(4):1–13, July 2013. ISSN 0730-0301, 1557-7368. doi: 10.1145/2461912.2462024. URL https://dl.acm.org/doi/10.1145/2461912.2462024.

[15] S. Laine and T. Karras. Efficient Sparse Voxel Octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, August 2011. ISSN 1077-2626. doi: 10.1109/TVCG.2010.240. URL http://ieeexplore.ieee.org/document/5620900/.

[16] Donald Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, June 1982. ISSN 0146664X. doi: 10.1016/0146-664X(82)90104-6. URL https://linkinghub.elsevier.com/retrieve/pii/0146664X82901046.

[17] Xing Mei, Philippe Decaudin, and Bao-Gang Hu. Fast Hydraulic Erosion Simulation and Visualization on GPU. In *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, pages 47–56, Maui, HI, USA, October 2007. IEEE. ISBN 978-0-7695-3009-3. doi: 10.1109/PG.2007.15. URL http://ieeexplore.ieee.org/document/4392715/.

[18] J. J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703–1759, August 2005. ISSN 0034-4885, 1361-6633. doi: 10.1088/0034-4885/68/8/R01. URL https://iopscience.iop.org/article/10.1088/0034-4885/68/8/R01.

[19] B. Neidhold, M. Wacker, and O. Deussen. Interactive physically based Fluid and Erosion Simulation. *Eurographics Workshop on Natural Phenomena*, page 8 pages, 2005. ISSN 1816-0867. doi: 10.2312/NPH/NPH05/025-032. URL http://diglib.eg.org/handle/10.2312/NPH.NPH05.025-032. Artwork Size: 8 pages ISBN: 9783905673296 Publisher: The Eurographics Association.

[20] J.F. O'Brien and J.K. Hodgins. Dynamic simulation of splashing fluids. In *Proceedings Computer Animation'95*, pages 198–205,, Geneva, Switzerland, 1995. IEEE Comput. Soc. Press. ISBN 978-0-8186-7062-6. doi: 10.1109/CA.1995.393532. URL http://ieeexplore.ieee.org/document/393532/.

[21] A. Peytavie, E. Galin, J. Grosjean, and S. Merillou. Arches: a Framework for Modeling Complex Terrains. *Computer Graphics Forum*, 28(2):457–467, April 2009. ISSN 01677055, 14678659. doi: 10.1111/j.1467-8659.2009.01385.x. URL http://doi.wiley.com/10.1111/j.1467-8659.2009.01385.x.

[22] R. S. Sarracino, G. Prasad, and M. Hoohlo. A mathematical model of spheroidal weathering. *Mathematical Geology*, 19(4):269–289, May 1987. ISSN 0882-8121, 1573-8868. doi: 10.1007/BF00897839. URL http://link.springer.com/10.1007/BF00897839.

[23] Ya-Lun Zeng, Charlie Irawan Tan, Wen-Kai Tai, Mau-Tsuen Yang, Cheng-Chin Chiang, and Chin-Chen Chang. A momentum-based deformation system for granular material. *Computer Animation and Virtual Worlds*, 18(4-5):289–300, September 2007. ISSN 15464261, 1546427X. doi: 10.1002/cav.209. URL https://onlinelibrary.wiley.com/doi/10.1002/cav.209.