2011-11-30

Fast Spheroidal Weathering with Colluvium Deposition

McKay T. Farley
Brigham Young University - Provo

Fast Spheroidal Weathering with Colluvium Deposition

McKay Farley

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

December 2011

ABSTRACT

Fast Spheroidal Weathering with Colluvium Deposition

McKay Farley
Department of Computer Science, BYU
Master of Science

It can be difficult to quickly and easily create realistic sandstone terrain. Film makers often need to generate realistic terrain for establishing the setting of their film. Many methods have been created which address terrain generation. One such method is using heightmaps which encode height as a gray-value in a 2d image. Most terrain generation techniques don't admit concavities such as overhangs and arches. We present an algorithm that operates on a voxel grid for creating 3d terrain. Our algorithm uses curvature estimation to weather away the terrain. We speed up our method using a caching mechanism that stores the curvature estimate. We generate piles of colluvium, the broken away pieces of weathered rock, with a simple deposition algorithm to improve the realism of the terrain. We explore the possibility of generating our sandstone terrain on the GPU using OpenCL. With our algorithm, an artist is able to quickly and easily create 3d terrain with concavities and colluvium.

Keywords: Terrain Generation, Weathering, GPGPU

Table of Contents

Chapter 1

Introduction

This paper investigates the problem of generating 3D weathered terrain at interactive speeds. We generate terrain, like the rock columns in figure 1.1, using a geologically inspired algorithm.

Terrain is one of the tools which assists in establishing the setting of a story. Having a good setting is a key element in effective story telling. Directors use terrain to evoke various emotions in their viewers. For a fantasy war scene, the setting may be in a foreboding valley with jagged rock cliffs surrounding the battle. A distant and unknown planet may be set in a dessert like plane with curious rock pillars scattered throughout. Computer generated terrain is now a common technique in realizing the director's vision.

1.1 Terrain Generation Methods

The most common technique in creating CG terrain is to use heightmaps. A heightmap is a two dimensional grid of values which represent the altitude, or z-value, of the terrain at the given (x, y) coordinates. By encoding the altitude of the terrain as the brightness of a pixel in a gray-scale image, we can generate a large variety of terrain like that in figure 1.2. Fractal patterns, such as fractional Brownian motion (fBm), are common for quickly and easily generating rocky mountain ranges (Mandelbrot [1982]). However, such fractals result in terrain that is just as rocky at the base of the terrain as it is at the peak. This is not very realistic since rocky mountains are generally smoother at lower altitudes. To add more realism, one could smooth the lower terrain by varying the frequency of the fBm based on

Figure 1.1: Thor's Hammer in Bryce Canyon, Utah

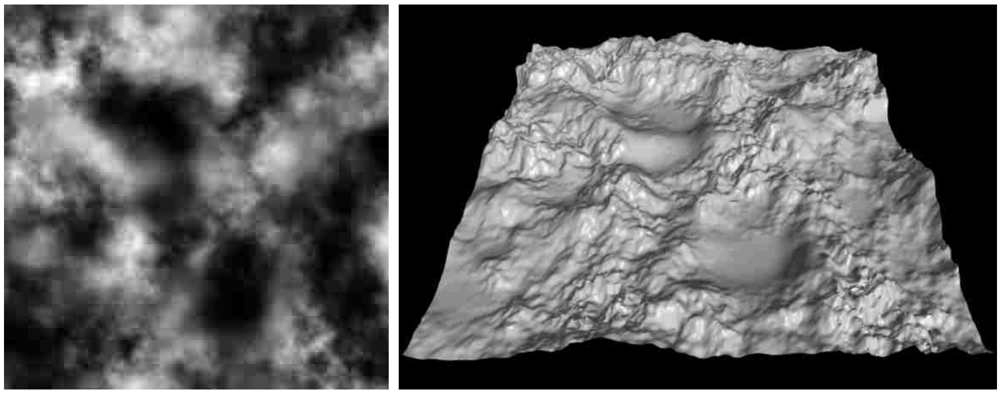
Figure 1.2: Example of a heightmap generated in Terragen next to a 3d render of the terrain

height (Musgrave et al. [1989]). Other methods for making the terrain more realistic can be done as a post process. One example is a more physical approach which simulates hydraulic erosion and thermal weathering (Musgrave et al. [1989]). Such a process moves terrain down from higher altitudes and causes the terrain to settle at lower altitudes. The result is written back to the heightmap.

1.2 Concave Terrain

The main limitation of heightmaps is that they can only encode a single altitude per pixel point. So terrain generated from heightmaps cannot admit concavities such as overhangs, arches and caves. The only way to generate concave terrain with heightmaps is to generate an initial form from the heightmap, and then manually add the concave features to the generated polygonal mesh. One could do so with sculpting tools that further subdivide the terrain's polygons and then push and pull vertices to get the desired shape (Dorsey et al. [1999]). Another way to generate more interesting terrain is through sediment deposition and fluid flow. By simulating the flow of water over the terrain, we can simulate hydraulic erosion which can eat away at terrain walls and generate overhangs (Ito et al. [2003]). Both of these methods move away from the heightmap representation and cannot store the result as a heightmap as the former process does.

We present a phenomenological algorithm for generating natural looking sandstone terrain with concave features. Our approach is sped up through caching and generates a more natural look through colluvium deposition. Colluvium is the piles of sand and rock found sloping away from cliffs and rock columns. Since graphics hardware companies have opened up their hardware for developers to utilize their massive computing power, we also discuss the possibility to speed up our process using the GPU.

Chapter 2

Related Work

For years terrain generation techniques have involved using fractals as input to heightmaps, and then operating on those heightmaps. Heightmaps allow us to specify a height for each index in a two dimensional array. Mandelbrot observed the fractal nature of terrain and used them to generate heightmap terrain (Mandelbrot [1982]). Fournier et al. [1982] simplify the fractal approach using random midpoint displacement to create an approximation to fractional Brownian motion. Musgrave operates on fractal heightmaps to smooth the terrain using thermal weathering, which produces talus slopes, and hydraulic weathering (Musgrave et al. [1989]). However, due to limitations of heightmaps, such as their inability to represent overhangs and arches, recent work has been done using other representations which admit concave features.

Weathering is our main approach to terrain generation. Dorsey et al. [1999] developed a system for weathering stone statues and carvings. Dorsey uses a voxel discretization of the model surface and simulates liquid penetration and mineral deposition in that space. Chen's γ-tons simulate weathering by traversing model space using a process based on ray tracing (Chen et al. [2005]). Both methods are quite general and allow arbitrary geometry but are slow.

Advances in GPU technology could reduce the time required to simulate weathering. General purpose languages for the GPU simplify the process of implementing algorithms for the GPU. With these advances, large scale dynamics and simulations are becoming more feasible. Šťáva et al. [2008] implemented shallow water hydraulic erosion and deposition on

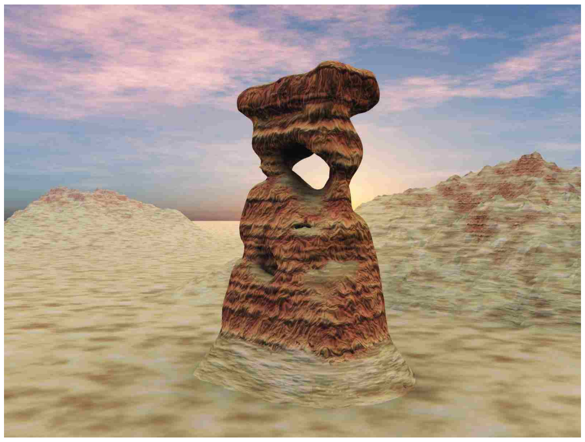

Figure 2.1: Example of weathering through curvature estimation in Jones et al. [2010]

the GPU. Their work allows interactive modeling of large scale terrain and real-time results, but uses a heightmap representation. Their other contribution of allowing different material layers through layering multiple heightmaps would be ideal for terrain like that found in Bryce Canyon National Park and Goblin Valley State Park in Utah.

Recent advances have been made that admit concave features in terrain. Ito et al. [2003] used a voxel representation of terrain and inserted joints to allow weathering. Beneš et al. [2006] used a run-length encoded voxel grid to simulate deposition of materials through hydraulic erosion.

One of the many concave terrain features found in nature is that present in Goblin Valley State Park, Utah. Milligan observed that the rock faces with a high surface area to volume ratio weathered faster, resulting in hard edges and corners becoming smooth (Milligan [2003]). This process, which he calls spheroidal weathering, was the basis for the virtual weathering of rock done by Beardall et al. [2007], which simulated spheroidal weathering on a discrete voxel grid. We improved the initial spheroidal weathering algorithm by adding decimation caching, such that instead of calculating the decimation rate at each simulation step it's able to lookup the value and update only when necessary. Later work done by Jones, extended the spheroidal weathering work, adding directability by allowing parameters, such as resistance to weathering and bubble type, to be modified interactively (Jones et al.

[2010]). Jones observed that the spheroidal weathering acts upon areas of high mean curvature and added cavernous weathering which weathers areas of negative mean curvature. The decimation caching method inspired and facilitated the use of negative curvature to generate cavernous weathering. In addition to directablility and cavernous weathering we added colluvium deposition to get a more realistic result from the weathering process. Vicsek [1984] and Pimienta et al. [1992] also developed algorithms for estimating curvature based on counting pixels in a neighborhood, but for different purposes than Jones's voxel counting. Pimienta showed that the average curvature estimation of a surface is linearly related to the actual curvature.

We will discuss our work involved in enhancing the algorithms created by Beardall et al. [2007] and Jones et al. [2010]. We speed up the spheroidal weathering algorithm by caching the weathering rate for each voxel. And we improve the natural look and feel of the terrain by adding colluvium deposition.

Chapter 3

Weathering

3.1 Spheroidal Weathering

The main part of our terrain generation method is spheroidal weathering. Spheroidal weathering, coined by Milligan [2003], is the process by which hard edges become smooth. Points on a rock with high mean curvature will weather faster than points with low mean curvature. Figure 3.1 shows the mean curvature for different points in a cube. The mean curvature is proportional to the weathering rate of a point on the surface. Red areas weather faster and green areas weather slower. The corners weather faster than the edges because they are exposed to more air and to the elements of nature. The edges weather faster than the faces for the same reason, but they don't weather as fast as the corners because they're not as exposed. We estimate the mean curvature by centering a sphere around a voxel so that the more air there is inside the sphere, the higher the mean curvature.

To illustrate this further, figure 3.2 shows two points on the border of a square with different amounts of air exposure. If we center a circle around each point and calculate the amount of air within the circle, we see that the corner has more air surrounding it than the edge. Over time, the areas with more exposure weather faster than other areas. This results in a smooth shape whose surface points are all equally exposed. This process is the basis for our algorithm. In dealing with a 3D volume, we look at the air surrounding each surface point, or voxel, within a sphere. The mean curvature is proportional to the air to volume ratio around the voxel.

One more factor that needs addressing is when a rock contains different materials.

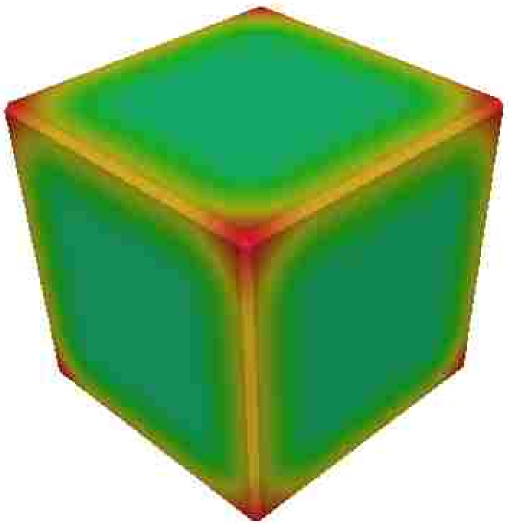

Figure 3.1: Color-coded weathering rates for the different parts of a cube

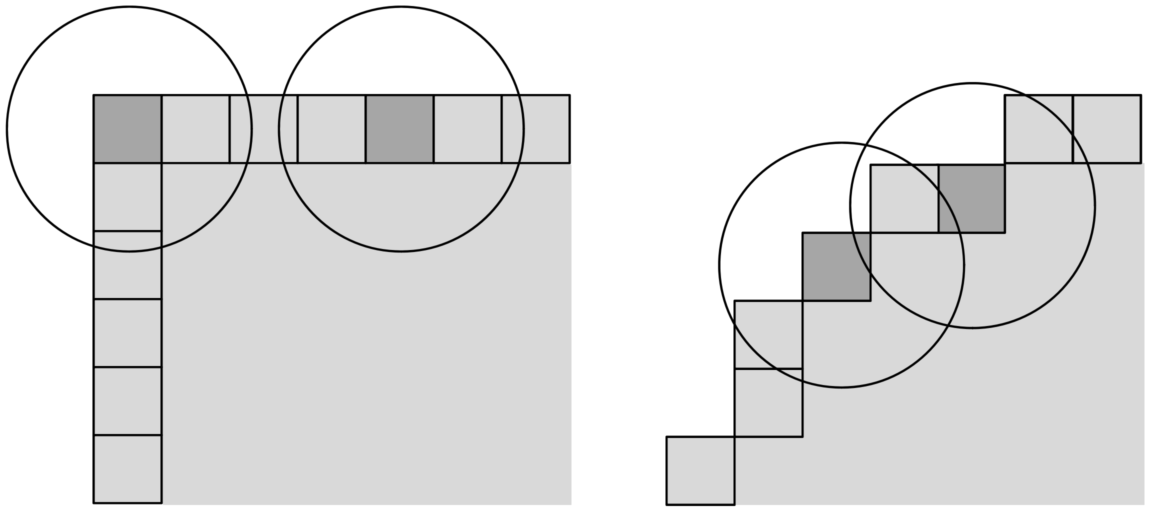

Figure 3.2: Two dimensional example of weathering rate due to exposure

Some materials may be more resistant than others to weathering. The voxels which are less resistant to weathering would weather faster than those that are more resistant. Therefore, the decimation rate for a voxel is further multiplied by a factor that is inversely proportional to the durability of the material contained within the voxel.

Our naive spheroidal weathering algorithm is a two step process. Each voxel stores the amount of rock contained in the voxel, which we call the current decimation value v_d. This is a number between 0 and the maximum decimation constant max_d. The first step in the process calculates a new decimation value, v_{dnew}, for each voxel. This is done by first calculating the air to volume ratio around the voxel by counting the number of air voxels V_{air} contained within a bubble b centered at the voxel and dividing it by the volume of the bubble b_{vol}.

$$ratio = \frac{V_{air}}{b_{vol}} \tag{3.1}$$

We then take the ratio and multiply it by max_d and subtract it from v_d.

$$v_{dnew} = v_d - ratio * max_d \tag{3.2}$$

This way more air voxels results in subtracting more rock and less air results in subtracting less.

To account for differences in resistance to weathering, each voxel has a durability, v_{dur}, associated with it. We take v_{dur} and calculate a percentage out of the maximum durability, max_{dur}, and use it to reduce the weathering rate. Since voxels of higher durability weather less than those of lower durability, we subtract the percentage from 1.

$$percent = 1 - \frac{v_{dur}}{max_{dur}} \tag{3.3}$$

Thus our final equation for calculating the new decimation rate is:

$$v_{dnew} = v_d - ratio * max_d * percent \tag{3.4}$$

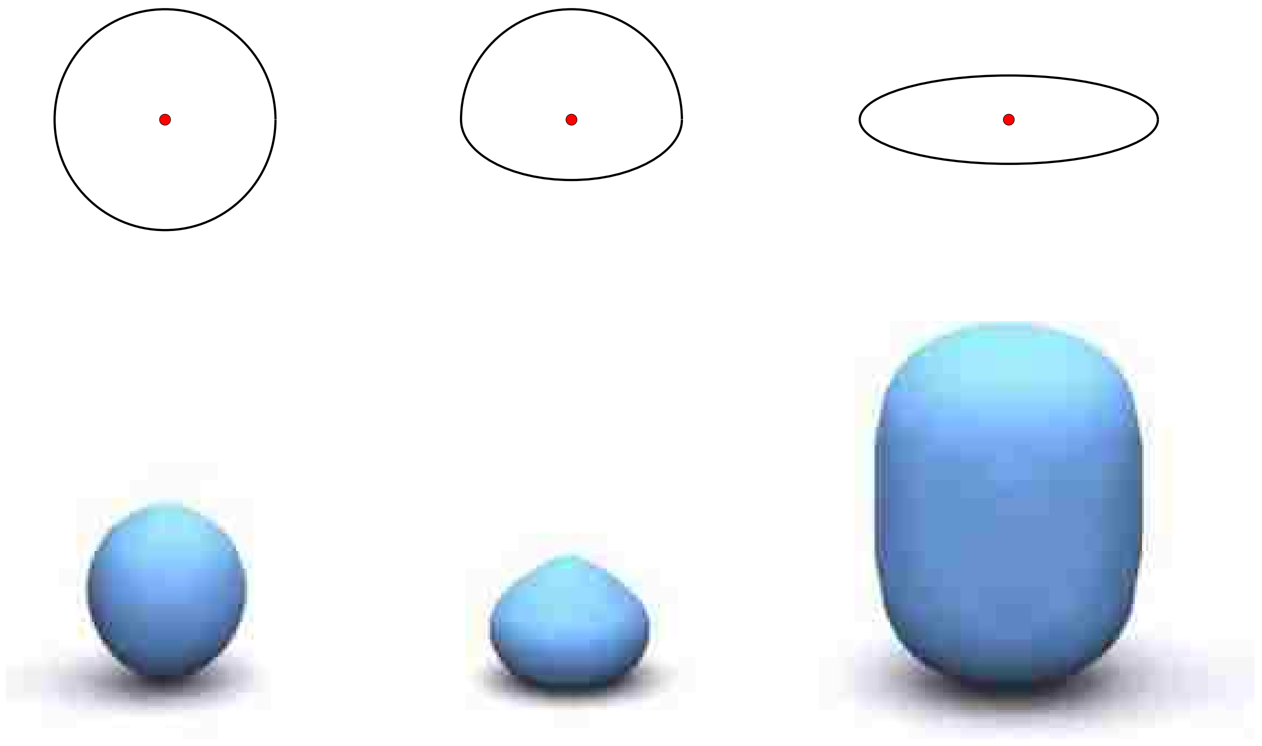

Figure 3.3: Different shaped bubbles produce different shaped terrain

The second step in our process sets the current value to the calculated new value. This two step process is required because the rate at which something weathers is dependent on the air to volume ratio contained in the bubble. Calculating the new decimation value for each voxel is the most expensive part of our simulation. For each voxel we visit all the neighbors in the neighborhood. For a bubble with diameter m, we would be visiting m x m x m voxels. Therefore, with a terrain of n voxels and bubbles of diameter m, we end up with a complexity of $O(nm^3)$ for each simulation step.

3.2 Bubble Type

One method of controlling the shape of the terrain during weathering is by using a different shape for the volume, or bubble, around each voxel. Figure 3.3 shows various bubble shapes centered around voxels (in red) above the rock shape they produce. The simplest form uses a sphere, which results in evenly rounded terrain (see left shape in figure 3.3). We observe that terrain isn't evenly rounded, but seems to weather more on top than on bottom. If we use a bubble of this shape that has more volume on top than on bottom, we get the desired shape (see center of figure 3.3). Such a bubble shape would underestimate the surface's curvature if it is on the bottom of the terrain. Therefore, voxels on the top of the terrain would weather faster than voxels on the bottom. If we want to weather around the voxel more than above and below it, we can use a disc shaped volume (right of figure 3.3). This would underestimate the curvature for voxels in the middle of the terrain, so they weather faster than the voxels on top and bottom.

3.3 Decimation Caching

We are able to speed up the algorithm by caching the number of air voxels in the bubble around each voxel and updating the voxel value at each step with that saved value. We can store in a table the rate for any number of air voxels contained in a bubble such that the index into the table is the number of air voxels contained in the bubble, resulting in:

$$decim\_rate(i) = \frac{i}{b_{vol}} * max_d \tag{3.5}$$

Since there can be different types of bubbles, this is a function of the bubble. When a voxel is initialized we count the number of air voxels surrounding it and save that value, v_a, with the new voxel. Finally, at each simulation step each voxel can simply lookup its decimation rate for its bubble using its saved air count.

$$v_{dnew} = v_d - b.decim\_rate(v_a) * percent \tag{3.6}$$

Thus we do not incur the cost of visiting neighbor voxels at each simulation step. We incur an initial cost of $O(nm^3)$ for initializing the voxels, but each simulation step now has a

```
1   foreach voxel decim in list_of_fully_decimated_voxels
2       foreach voxel neighbor in decim.neighborhood
3           if neighbor.bubble.contains(decim)
4           then ++neighbor.air_voxel_count
```

Figure 3.4: Neighbor update step

complexity of $O(n)$.

Since the number of air voxels surrounding each voxel changes as neighboring voxels are decimated, we still need to do a two step process for each simulation step. The second step involves updating the cached air voxel count for neighboring voxels when a voxel is decimated (see listing in figure 3.4). When a voxel v_{decim} is decimated, we need to visit all neighbors v_{nbr} surrounding that voxel (line 2) that could contain v_{decim} and update accordingly. The radius of the neighborhood is as large as the largest bubble radius, r_{max}, in the simulation. Therefore, the neighborhood is defined as the range between $v_{(x-r_{max},y-r_{max},z-r_{max})}$ and $v_{(x+r_{max},y+r_{max},z+r_{max})}$ around $v_{decim(x,y,z)}$. Each v_{nbr} whose bubble contains v_{decim} (line 3) will increment its air voxel count (line 4). Now, instead of visiting the neighbors of each voxel at each simulation step for counting air voxels, we visit neighbors twice: when a voxel is initialized and when it is fully decimated.

3.4 Colluvium

We add colluvium to help our algorithm generate more realistic results. As the elements weather away at rocks, pieces of rock are deposited at the base of the weathered rock. This piled up sediment is referred to as colluvium. An example of colluvium in nature is in figure 3.5, where the colluvium is outlined in red.

Our terrain weathering simulation mimics this process as voxels are decimated. A naive algorithm would simply delete a voxel when it is fully decimated. As our terrain weathers and voxels become fully decimated, instead of deleting the voxel we deposit it at the base of the terrain. The colluvial voxels are affected by the simulation step just as normal

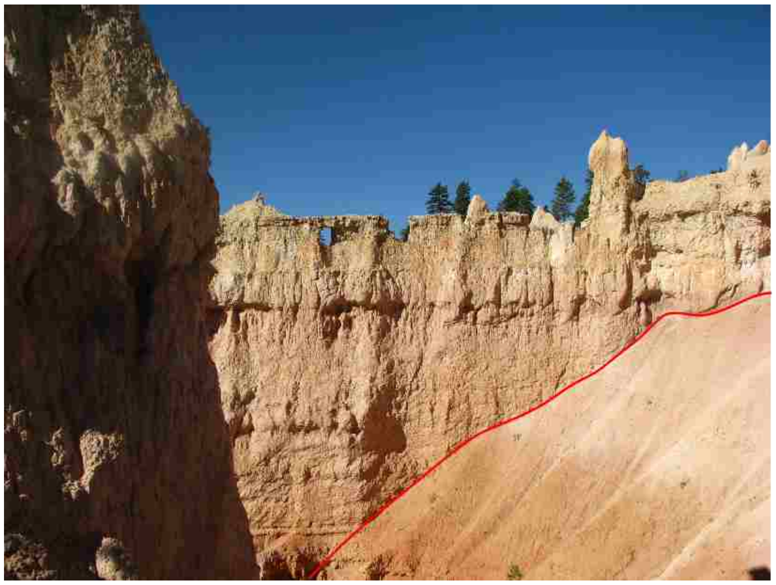
Figure 3.5: Example of colluvium slopes from sediment deposition

rock voxels, with the difference being that colluvial voxels are deleted when fully decimated.

Another approach to generating more realistic terrain is to fake colluvium. Since colluvium creates a slope of sediment at the base of the terrain, we can play with the durabilities at the base of the terrain such that the weathered result is sloped. We do this by putting thin layers at the bottom of the layer stack, and set high durabilties for the lower layers and gradually decrease the durability moving up the stack. This results in the upper layers weathering faster than the lower layers, generating a sloping effect similar to colluvium piles. Unfortunately, this is not as realistic as dropping the voxels as bits of rock fall in nature and doesn't create as natural a look.

3.4.1 Colluvium Deposition

Colluvium deposition happens once a voxel is fully decimated. In order to determine where to deposit the colluvium we must "drop" the decimated voxel (see listing in figure 3.7). When a voxel is fully decimated, v_D, it searches down in z, assuming z is up, for a voxel that still contains rock (line 2). Once it finds one at $v_{(x,y,z)}$ it then searches down the neighboring columns in x + 1, x - 1, y + 1, and y - 1 for a voxel below z (line 10). We exclude the

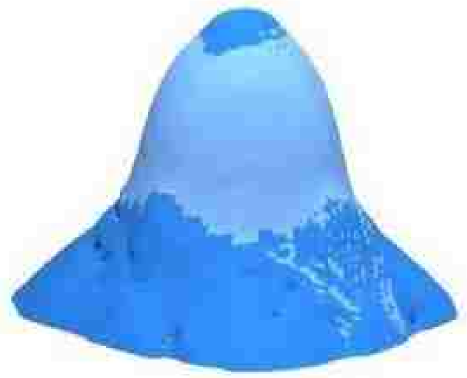

Figure 3.6: Simple rock column with colluvium deposited at its base

diagonal to save on processing time and because a voxel could move diagonally by falling in an x direction and then subsequently falling in a y direction. It then moves into the column containing the lowest un-decimated voxel $v_{(x',y',z')}$ and continues the search (line 14).

In nature colluvium deposition stops when the sediment reaches the angle of repose. Our implementation stops dropping a voxel when it reaches a predefined angle of repose. We calculate the angle of repose by the position of the falling voxel and the positions of the voxels in the below z in the four-connected neighboring columns. When calculating the angle of repose for voxel, v, and a neighboring voxel, nbr, we use the z value for each position in equation 3.7 (line 23).

$$\theta = tan^{-1}(v_z - nbr_z) \tag{3.7}$$

If nbr_z is less than v_z, and θ is less than or equal to the angle of repose, we have found our resting position for the voxel. For an angle of 45°, this is as simple as finding a difference of 1 for the z values. When it reaches the angle of repose, we initialize a new voxel at $v_{(x',y',z'+1)}$ and flag it as colluvium (line 15). Since the neighboring voxels now have a new voxel in their bubbles, we visit all neighbors to the new colluvium voxel and decrement their air count. Figure 3.6 shows an example of the final shape for colluvium deposition. The colluvium is colored dark blue and the normal rock is colored light blue.

```
foreach voxel v in fully_decimated_voxels
    below = findLowestVoxel(v.z, in v.column)
    new_voxel = drop(voxel(below.z+1))
    delete v
    new_voxel.init()
    new_voxel.colluvium = true
    new_voxel.decrementNeighbors()

function drop(voxel v)
    foreach column c in [col(v.x-1), col(v.y-1), col(v.x+1), col(v.y+1)]
        nbr = findLowestVoxel(v.z, in c)
        if nbr.z >= v.z then continue
        angle = calcAngle(v, nbr)
        if angle > angle_of_repose then return drop(voxel(nbr.z+1))
    return new voxel(v.pos)

function findLowestVoxel(z, column)
    foreach voxel v in column.voxels(from z, to 0)
        if v is not air then return v
    return limit_voxel #identifies lower boundary of simulation space

function calcAngle(v, nbr)
    return arctan(v.z - nbr.z)
```

Figure 3.7: Colluvium deposition algorithm

Chapter 4

Weathering on the GPU

In this chapter we discuss how we implemented our spheroidal weathering algorithm on the GPU. We offer a brief introduction to the OpenCL architecture. Then we describe how we represent the terrain on the GPU. Finally, we explain the implementation details for decimation caching and colluvium shaping for our GPU implementation.

4.1 Architecture

We wrote our GPU implementation in OpenCL. OpenCL is a framework for parallel computing on CPUs and GPUs. It allows us to utilize the massively parallel processing cores on today's graphics cards.

OpenCL is a data driven model and it distributes the work among work-groups by dividing the data to be worked on. Each work-group, in turn, is divided into work-items, which have shared memory within the work-group. Figure 4.1 shows an example of this with a two dimensional dataset. Each work-item executes a kernel. A kernel is a function which runs on the device and is called from the host. There is no communication between work-groups.

OpenCL provides multiple memory spaces (see figure 4.2), each with their own benefits and drawbacks. The first memory space is global memory. Global memory is accessible to all work-items but can be slow to access. In the case of graphics cards global memory is all of the memory on the card. Next we have local memory, which is local to a work-group and is shared between the work-items in the work-group. The work-items are grouped physically

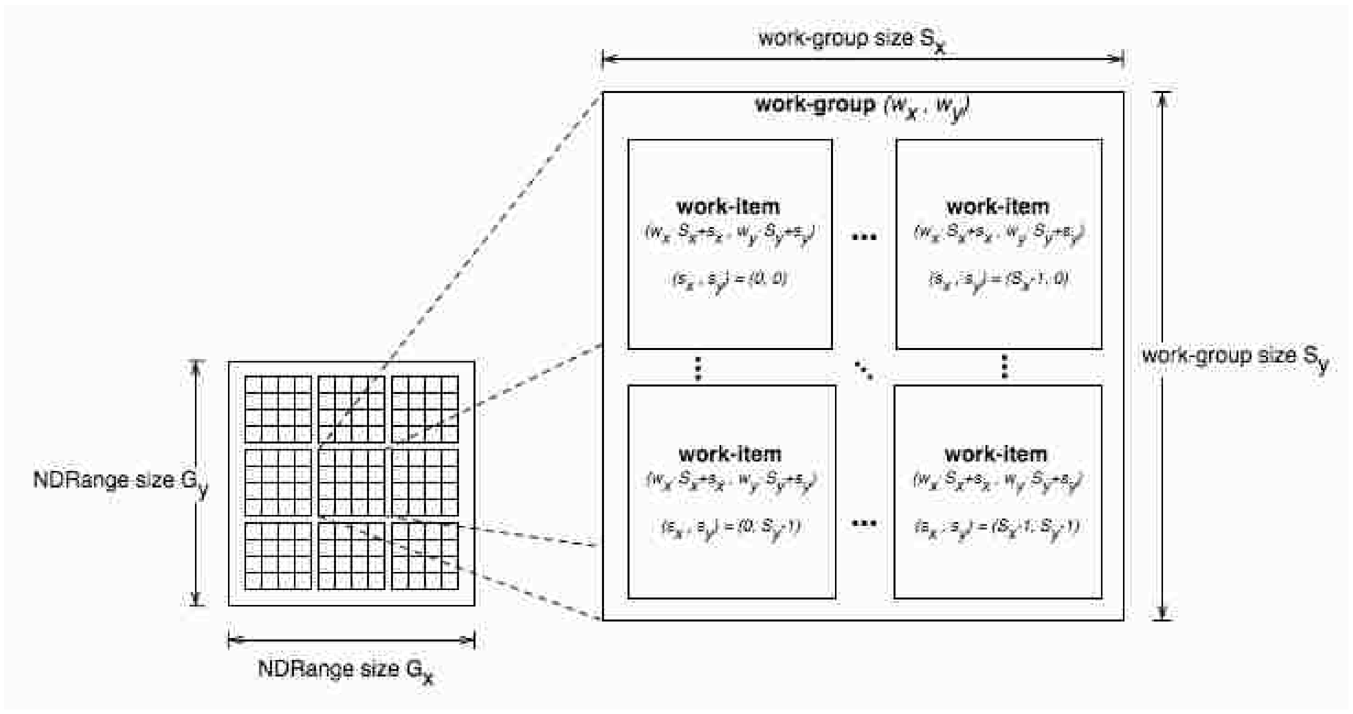
work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x S_x+s_x, w_y S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x S_x+s_x, w_y S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-group size S_y
work-item
$(w_x S_x+s_x, w_y S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x S_x+s_x, w_y S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
NDRange size G_y
NDRange size G_x

Figure 4.1: Example of OpenCL's organization of work (Munshi [2009])

on a compute unit. Access to local memory can be synchronized between work-items and is faster to access than global memory. The next memory space down the hierarchy is private memory. Private memory is local to each work-item which runs on a processing element (PE) and is the fastest to access. Constant memory doesn't fit in the hierarchy but is accessible to all work-items, is immutable, and is faster to access than global memory.

The OpenCL architecture is single-instruction, multiple-data (SIMD). What makes parallel computing on the GPU really fast is when all the cores are doing the same thing but on different values. The cores in a processing element share a program counter. If the instruction to be executed on each core is the same for each step, then we have the ideal situation. However, if there is any branch in code paths that causes two cores to process differently, then it will execute one and then the other instead of in parallel. Therefore, the more branching there is the longer the compute time.

For example in figure 4.3 we have 4 cores operating in parallel. Each instruction is

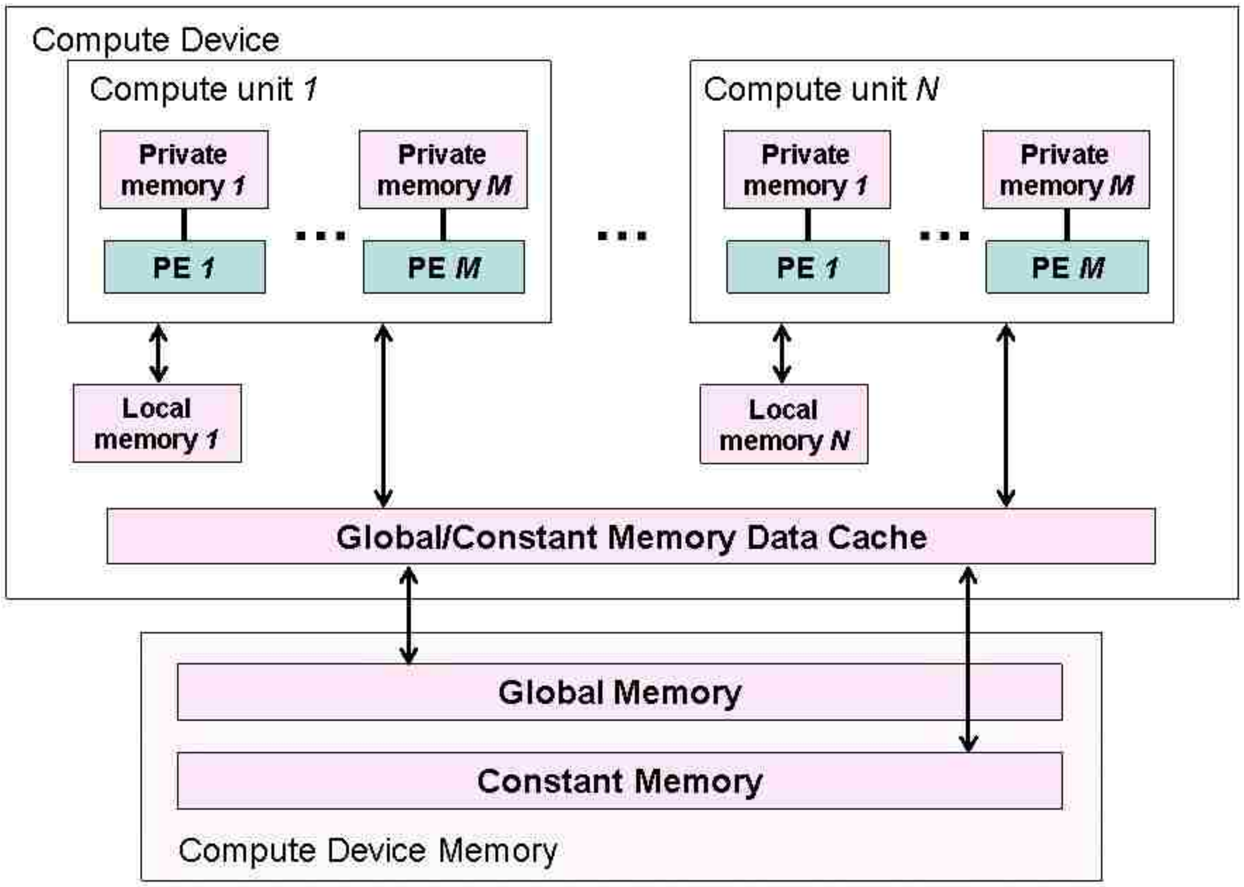

Figure 4.2: OpenCL memory model (Munshi [2009])

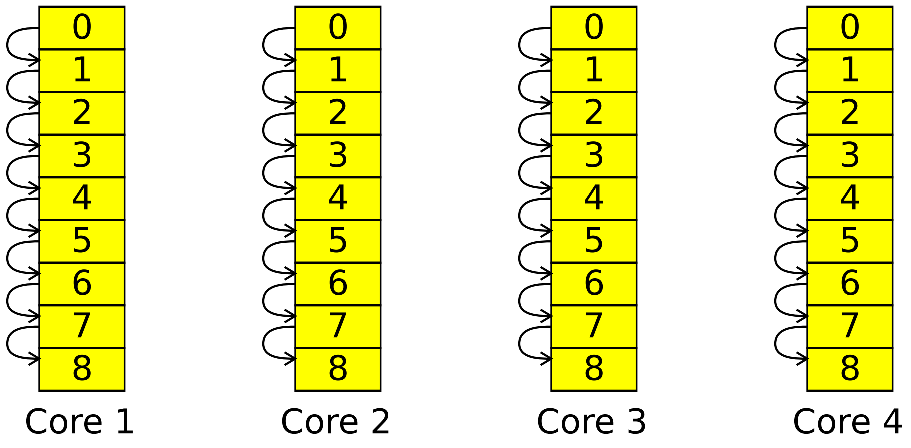

Figure 4.3: Ideal code path parallelization for SIMD in which all cores execute the same set of instructions

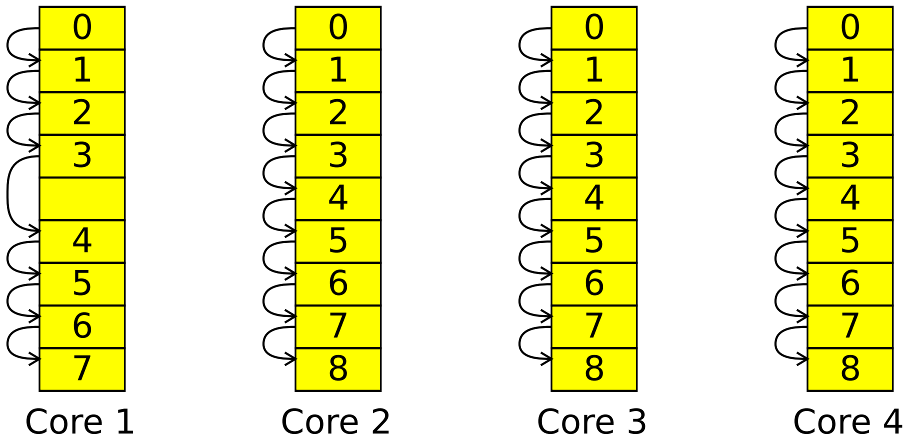

Figure 4.4: Code path with one core executing a different set of instructions

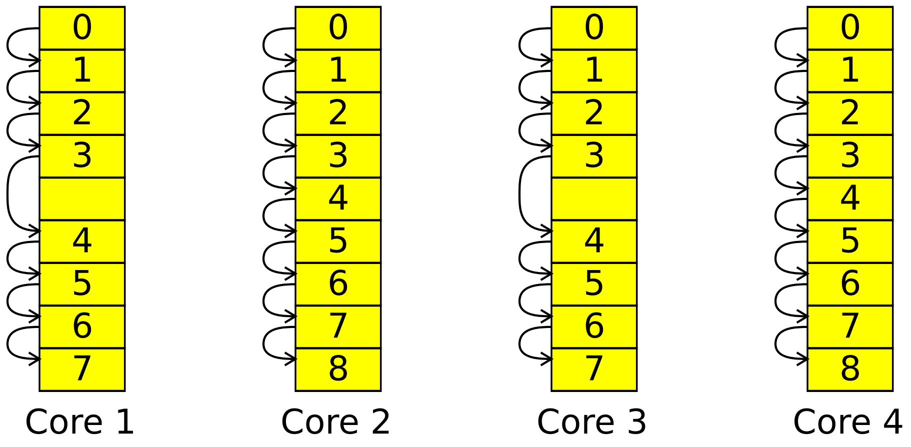

Figure 4.5: Code paths in which an equal number of cores are executing separate instructions

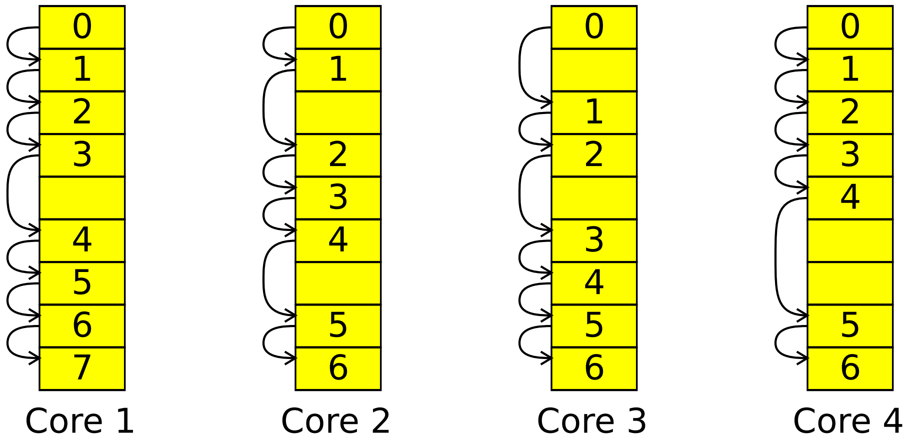

Figure 4.6: Poor code paths for SIMD parallelization in which all cores execute different instructions

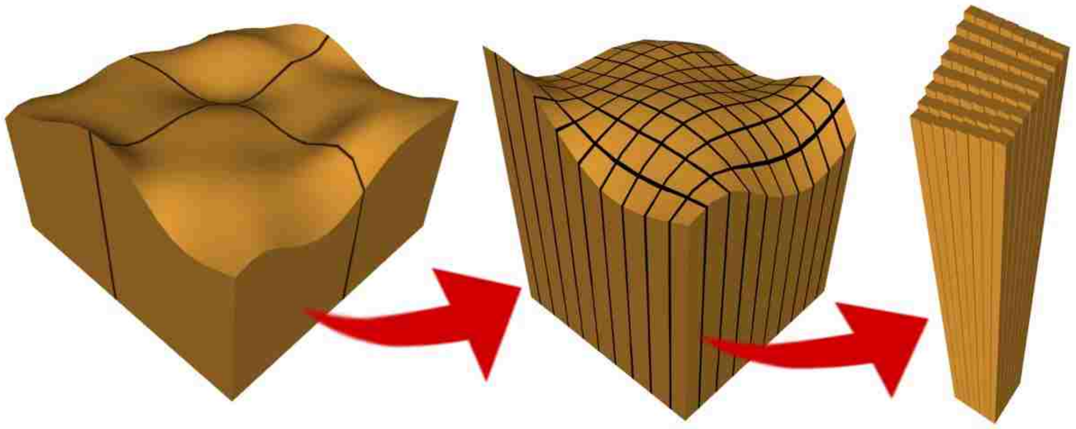
Figure 4.7: Grid division levels

executed at the same time on each core. This is the ideal setup for OpenCL computing. If we have a single core skip an instruction, like in figure 4.4, then for step 4 it will execute the instruction for core 1 and then the instructions for the other cores. Now that the first core is out of sync with the other cores for the rest of the steps, the instruction in the first core is executed before the instructions in the other cores, effectively doubling the execution steps. This theoretically doubles the compute time, but the slow down is actually larger due to switching between the two contexts for core 1 and cores 2, 3, and 4. If another core also skips an instruction but skips the same instruction as the first core, as in figure 4.5, we have the same situation because it will execute the same as in figure 4.4 with two contexts. The worst case scenario would involve all the cores executing different instructions at each step, similar to figure 4.6. We would end up with 4 different instructions being executed at each step, in addition to 4 context switches. This situation drastically slows down performance and makes it worse than if the program were run in serial.

4.2 GPU Terrain Model

OpenCL requires us to specify how to divide up the terrain so it can correctly divide up the work between work-items and work-groups. We divide up the terrain similar to how archaeologists divide a dig site into a grid. In order to divide the terrain among the work groups we split it up into sections, as shown in the left side of figure 4.7, which are assigned a work-group. Each section is then divided amongst the work-items in that group as shown in the center of figure 4.7. Each work-item processes multiple columns as shown in the right side of figure 4.7. This method groups columns of discrete voxels, dividing the terrain along the longitude and latitude. The terrain is stored in global memory and is kept there due to the possible size of data given to each work-item.

Each voxel lies in a layer which has a material associated with it. The material stores properties such as durability (or resistance to weathering), noise coefficients, and the bubble type. The layer table is a run-length encoded array which stores the material for each layer and how many voxels thick the layer is. Due to the possible size of the layer table, we store it in global memory, but the material lookup is stored in constant memory.

4.3 Decimation Caching

Caching the decimation rate makes the CPU algorithm really fast. We implemented caching for the GPU algorithm in an attempt to get the same speed benefit. The terrain is stored on the host as one contiguous array of integers that represent the decimation value of each voxel. This representation makes it easier to copy the data to and from the GPU. When we initialize the terrain, we take an initial pass over the data and determine the decimation rate of each voxel just as we do in the CPU version. The decimation rate is stored in a second contiguous array that is parallel to the decimation value array. OpenCL requires that we copy the data from the host to the device, but the time to copy is negligible.

The rest of the simulation is broken up into multiple steps, each of which is implemented as a separate kernel. The first step updates the decimation value for each voxel based on the decimation rate. This decimation step requires a material lookup because the decimation rate is associated with the bubble. The next step in the simulation updates the decimation rate of neighbors of voxels which have been fully decimated. The neighbor update is dependent on the new decimation value from the previous step. This requires all the work-items and work-groups to be done computing the decimation update kernel before executing the neighbor update. Since there is no communication between work-groups, we implemented this as a separate kernel and we synchronize the work-groups on the host with `clFinish`.

Finally, we extract a preview surface using the marching cubes algorithm implemented on the GPU. Marching cubes calculates an iso-surface by finding points between voxels and connecting them with triangles. The position of each point is a linear interpolation between voxel positions using the values stored in each voxel. Since the terrain data already consumes so much memory on the GPU, we need to save as much space as possible with the surface extraction. If we were to compute the interpolated point on the GPU, we would be storing 4 floats for each voxel pair. To reduce the amount of data used by surface extraction on the GPU, our marching cubes algorithm determines between which voxels the surface points lie and then the host calculates the interpolated position. Therefore, the GPU only needs to store 2 integers (the indices of the voxels) for each voxel pair that has a point between them.

4.4 Colluvium Deposition

Depositing colluvium, as described in section 3.4.1, requires moving data between terrain columns. This doesn't fit in the work-item/work-group architecture. As voxels are fully decimated and dropped, they enter neighboring columns which may be in a neighboring work-group. Since OpenCL does not allow communication between work-groups, we cannot move voxels between work-groups. However we can use the fake approach to colluvium by inserting thin layers of decreasing durability at the bottom of the terrain with the more durable layers at the base.

While this doesn't generate colluvium piles at the base of the terrain, it does generate sloping bases similar to talus slopes formed from colluvium. However, this increases the number of layers and thus increases the run-length encoded table, thereby slowing down material lookup.

Chapter 5

Results

In this chapter we present our results for our CPU and GPU spheroidal weathering algorithms. First we discuss the improvements to our CPU algorithm through decimation caching. Then we show the effect colluvium deposition has on the final shape. We compare different results for the GPU weathering algorithm with different terrain configurations. Finally, we compare the speed between the CPU and GPU algorithms.

5.1 Decimation Caching

Table 5.1: Average uncached vs cached time to compute weathering in seconds for a single simulation step

Voxel Count	Uncached	Cached	Uncached/Cached
30,691,011	2004.53	30.063	66.678
33,554,432	2062.31	151.818	13.584
33,909,595	2248.95	36.509	61.600

Without decimation caching, we would have to visit the neighbor of every rock voxel at each simulation step. Given that we do this now as an initialization step, we have an approximate time it would take to weather at each step without decimation caching. Table 5.1 shows the approximate savings we get at each simulation step for a given number of voxels on the CPU based on the initialization time. The second row takes longer for the cached version than the other two rows because there are more voxels fully decimated at each step and the

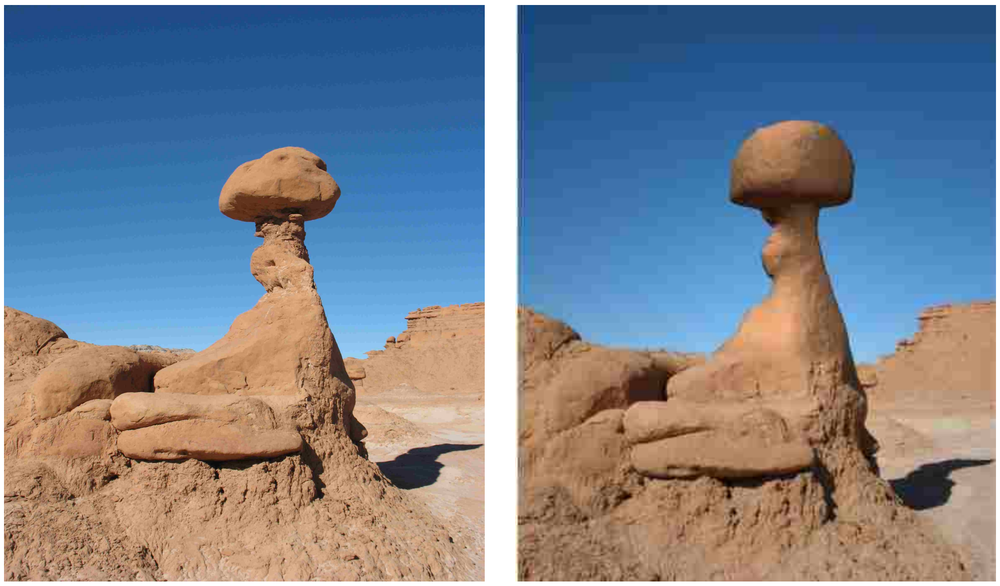
(a) Photo (b) Render over photo

Figure 5.1: Desired result of a rock with a sloped base next to a final rendered rock column

simulation is therefore spending more time per step updating neighbors. The first and third rows are closer in cached time because the configuration of the terrains are more similar. With caching we get as little as 13x and as much as 70x speed on the CPU.

5.2 Colluvium Deposition

Figure 3.6 shows a simple rock column with colluvium deposited around it. However, the purpose of adding colluvium is to make the terrain visually plausible. Figure 5.1(a) shows an existing rock column that we'd like to replicate. Given the right inputs of layer durability and noise (see figure 5.2(a)), we are able to generate the desired shape with colluvium deposited at the base of the column, as seen in figure 5.2(b). Figure 5.1(b) shows a final render of our rock column positioned in front of the reference photo.

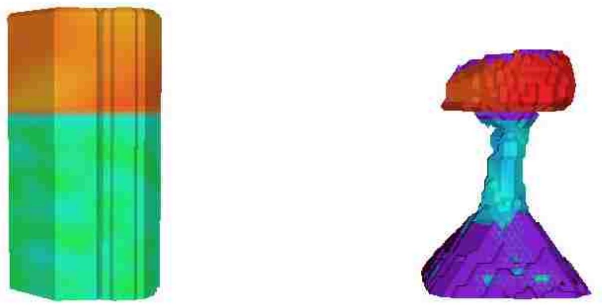
(a) Initial shape (b) Final shape

Figure 5.2: Initial and final shapes of the simulation (color shows resistance to weathering)

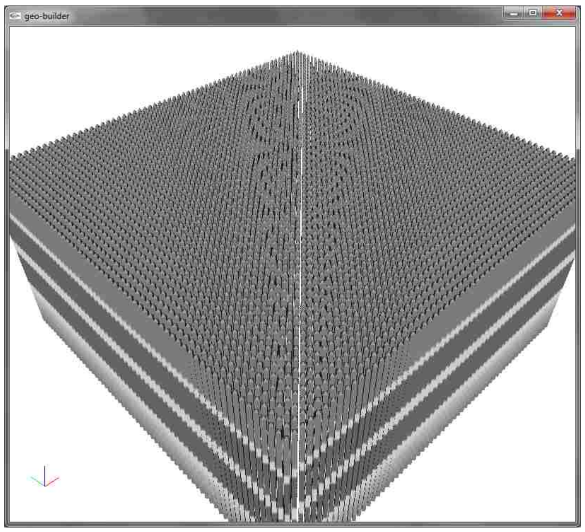

Figure 5.3: Terrain configuration catered to OpenCL limitations

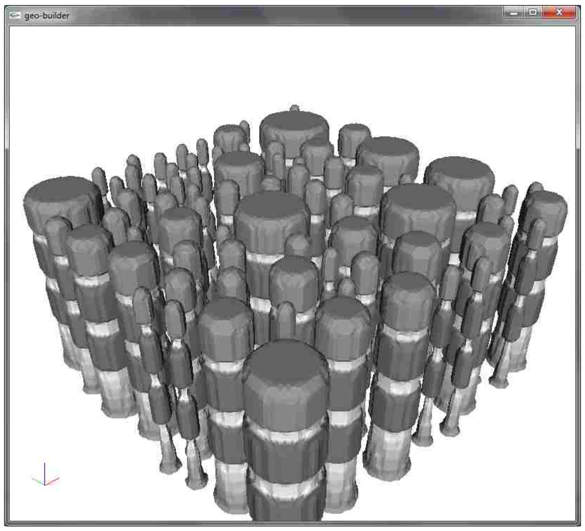

Figure 5.4: Goblins with layers of equal height

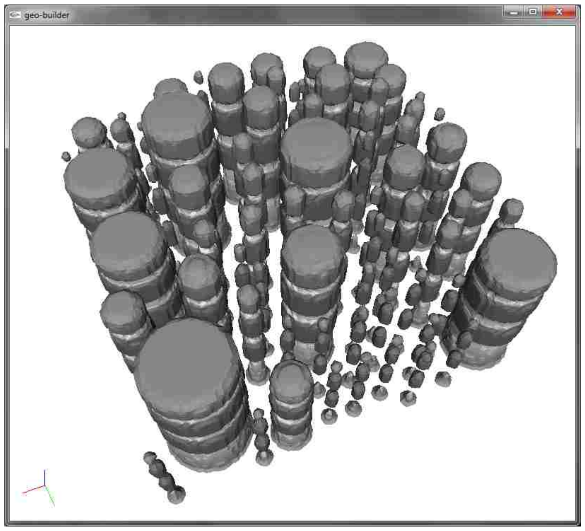

Figure 5.5: Goblins of varying height

5.3 GPU Method

Table 5.2: Total simulation time in seconds

Config	**CPU**	**GPU**	**CPU/GPU**
`Uniform`	6072.319	2437.762	2.491
`Noisy`	6588.288	2682.327	2.456
`Goblins`	5846.013	9921.501	0.589
`Varying`	5509.264	11136.012	0.495

Our GPU implementation is faster than the CPU implementation, but only sometimes. In the most common cases the CPU is faster than the GPU due to variation within the terrain. Variation hurts the efficiency of GPU algorithms due to the SIMD nature of the architecture. Table 5.2 shows the total simulation time for different configurations. When the terrain is tailored to the limitations of OpenCL, the GPU is 2.491x faster than the CPU. This configuration, which we'll call `Uniform`, has rock columns that fit inside the dimensions assigned to a work-item so each work-item has the same number and shape of voxels to work on. All the layers in the rock columns are equal between the columns, and the resistance to weathering is uniform throughout each layer. This allows the work-items to get the same layers at the same time and all voxels will decimate at the same rate relative to each work-item. Figure 5.3 shows this configuration after initialization.

The `Noisy` configuration is equivalent to `Uniform`, but has noise in the material's durability. Adding that single variation causes the simulation to slow down, since the voxels decimate at differing rates relative to each work-item. That is, some voxels will decimate before voxels at the same position relative to other work-items. Once this happens some work-items will be processing voxels while others will be skipping empty voxels. It slows down to the point where it is 2.456x faster than the CPU.

Our next configuration, which we'll call `Goblins`, has circular rock columns of varying radii. Figure 5.4 shows this configuration after several simulation steps. And our final configuration, as seen in figure 5.5, is equivalent to `Goblins` except that the layer height varies between each goblin. We'll refer to this configuration as `Varying`.

As we add more variables to the terrain, the total simulation time slows down the GPU implementation. `Goblins` is slower than `Noisy` because of the varying rock columns. This causes some work-items to be processing columns of voxels, while others only have empty space. `Varying` is slower than `Goblins` due to the varying layer heights between rock columns. Not only will some work-items finish processing columns before others, but the material lookup for a voxel varies between work-items as well. Since the layer table is run-length encoded, the lookup for one voxel could return earlier than neighboring voxels based on the varying layer heights. The total simulation time is measured by the time it takes the smallest rock column to fully erode. The most drastic drop is in adding the goblins of varying radii, where the GPU slows down from 2.456x faster than the CPU to 0.589x faster (or 1.697x slower).

Table 5.3: Initialization time

Config	**CPU**	**GPU**	**CPU/GPU**
`Uniform`	2050.44	46.465	44.129
`Noisy`	2074.17	46.6445	44.468
`Goblins`	2248.95	101.1665	22.230
`Varying`	2004.53	81.9005	24.475

Table 5.4: Average weathering time

Config	**CPU**	**GPU**	**CPU/GPU**
`Uniform`	159.855	84.908	1.883
`Noisy`	143.780	84.464	1.702
`Goblins`	36.509	189.343	0.193
`Varying`	30.063	162.726	0.185

Table 5.5: Average extraction time

Config	**CPU**	**GPU**	**CPU/GPU**
`Uniform`	7.722	0.764	10.107
`Noisy`	6.690	0.871	7.681
`Goblins`	4.366	1.288	3.389
`Varying`	3.601	1.290	2.791

Breaking down the entire simulation we have (1) the time to initialize the terrain, and then multiple steps of alternating (2) erosion and (3) surface extraction. Initialization time, in table 5.3, is always faster on the GPU. It ranges from 22.23x and 44.468x faster than the CPU. Average weathering time, shown in table 5.4, is faster for the first two configurations (1.7-1.9x), since they are tailored to OpenCL's architecture, but significantly slower for the second two (5.1-5.5x), because of the varying rock column radii and layer heights. Average extraction time (table 5.5) is always faster but slows down for each additional variable, since voxels weather at different rates and also for the same reasons decimation slowed for `Goblins` and `Varying`, and ranges between 2.791x and 10.107x faster than the CPU.

All of our GPU/CPU comparison simulations are done on a terrain of dimensions 512x512x256. Our GPU algorithm was run on an NVidia Quadro FX 4800.

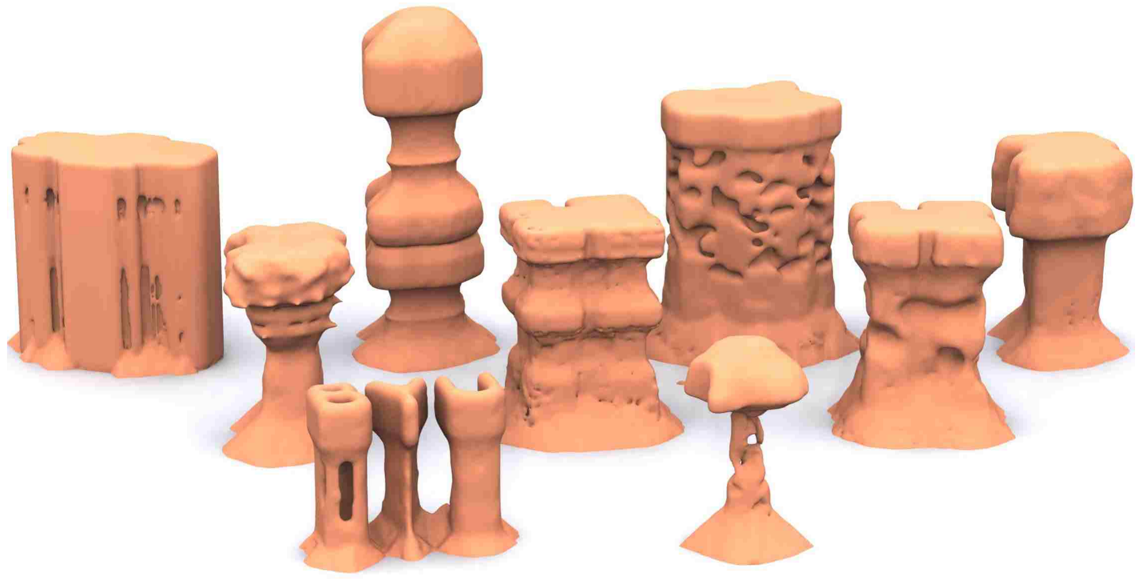

Figure 5.6: Variety of goblins generated through spheroidal weathering

5.4 Models and Renders

Our terrain generation techniques allow us to create a variety of realistic shapes. Once we have generated the geometry, we can export an isosurface in the OBJ file format. The geometry can then be imported into any 3D modeling package, that supports the format, and rendered to create beautiful images as seen in figures 5.6, 5.7, 5.8, and 5.9.

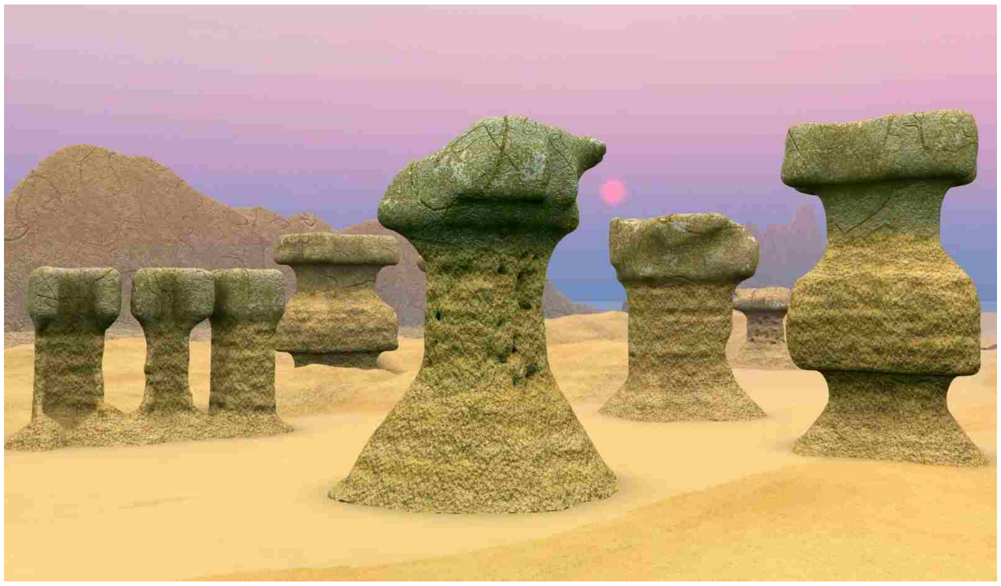
Figure 5.7: Group of goblins

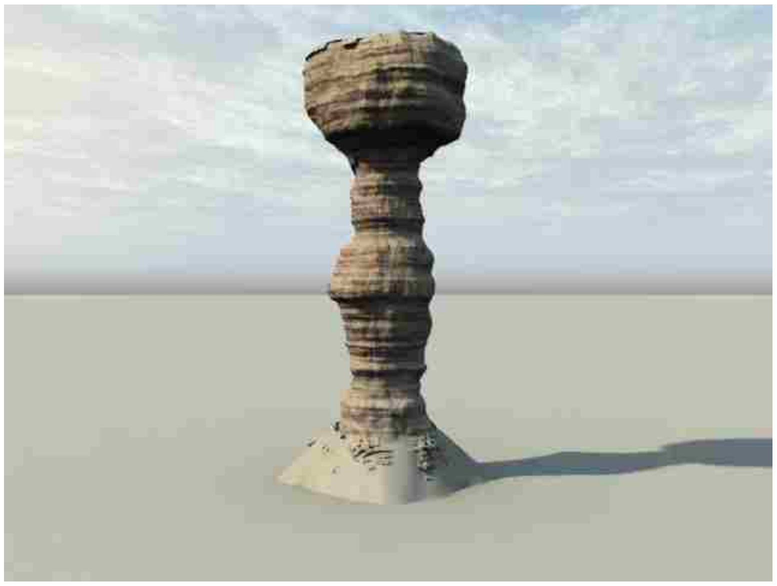
Figure 5.8: Single goblin generated through spheroidal weathering

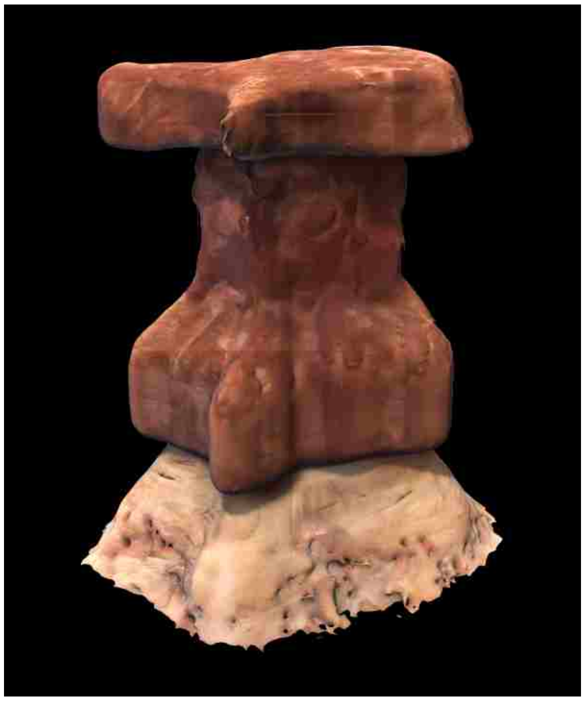

Figure 5.9: Another example of a single goblin

Chapter 6

Conclusions and Future Work

As explained in chapter 5, the area that needs the most improvement for the GPU implementation is the weathering step. It's the weathering step that slows down significantly as we add variation to our terrain. The reason for the slow down is the SIMD nature of the OpenCL architecture. As we add more variablility for each work-item, the code paths for each work-item branches more often. This is especially true for the neighbor update step. This step involves visiting all fully decimated voxels and updating their neighbors' decimation rate. During the neighbor update we visit every voxel and process only those that were fully decimated. Since not every voxel is fully decimated at the same time, we have excessive code branching.

Another reason for slow-down is the lookup into the run-length encoding for each layer's material, as seen in the slow-down between the `Goblins` and `Varying` configurations in section 5.3. When a work-item looks up the material for a voxel it must run-length decode the layer table. If there is variation in a layer's height, some work-items will return from decoding earlier than others. During the neighbor update step, each voxel neighbor visited must lookup the bubble for its material to know if the removed voxel was contained in it. Variation with a layer's height will cause branching here as well.

There could be multiple ways to improve our simulation for running on the GPU. For example, the neighbor update step could be compressed, such that only the fully decimated voxels get visited and processed. This would ensure that all the work-items are processing voxels instead of trying to find voxels to process. In order for this to be possible it would

Table 6.1: Total theoretical simulation time

Config	CPU	GPU	Hybrid
`Uniform`	6072.319	2437.762	3918.331
`Noisy`	6588.288	2682.327	4406.092
`Goblins`	5846.013	9921.501	3474.361
`Varying`	5509.264	11136.012	3387.881

require creating a list of voxels for the work-items to process and each work-item would process a portion of the list. The list would be of the voxel indices and would be padded to allow equal division between the work-items. This does run the risk that the last work-item would have less to do, but we're willing to accept that as long as the rest of the voxels are running in parallel.

Of course this improvement would require an additional fix that also addresses the run-length decoding slow-down. We would have to eliminate the decoding altogether so we don't take the decoding hit. The purpose of the run-length encoding is to save on memory usage. We could further save memory if we compress the material type into a word along with the decimation value or the decimation rate which are both stored in 4 byte *ints*. Storing the material index with the decimation value is the most likely candidate since the maximum decimation value is 255, which fits in a single byte. Then again, compressing it with the decimation rate is also feasible since the rate is proportional to the bubble's radius. We don't limit the radius, but a reasonable maximum radius of 10 voxels would only require 2 bytes to store the decimation rate, leaving 2 bytes for the material index. With this fix, material lookup would simply require indexing into the material table with the stored type, thereby reducing the amount of processing for each work-item during the neighbor update step. Not only does this improve the speed overall by eliminating the need to run-length decode the layer table, but it also avoids the branching slow-down from varying height within a layer.

These solutions are theoretical and might not work. In which case we would suggest using a hybrid approach which does everything on the GPU that is faster there and everything that is slower is done on the CPU. The only part of the simulation that is slowest is the neighbor update which can be done instead on the CPU. All other steps in the simulation can be done on the GPU. We would have to transfer data more between the host and device at each simulation step. After decimation, we would read the data back from the device to run the neighbor update. Then we would write the data back to the device for surface extraction. The extra data transfer would incur an additional overhead of 0.636 seconds on average with the graphics card we used. This would result in the theoretical times in table 6.1 under the Hybrid column. These numbers were calculated by taking out the CPU initialization time and the CPU surface extraction time and replacing it with the GPU times for each and adding the data transfer overhead. Given these caclulations we could have a solution that is significantly faster than the GPU in the common case (2.8-3.3x) and the CPU in all cases (1.5-1.7x).

References

M. Beardall, M. Farley, D. Ouderkirk, J. Smith, C. Rheimschussel, M. Jones, and P. Egbert. Goblins by spheroidal weathering. In *Proceedings of the Eurographics Workshop on Natural Phenomena*, pages 1–8, Prague, The Czech Republic, 2007. The Eurographics Association.

B. Beneš, V. Těšínský, J. Hornyš, and S. Bhatia. Hydraulic erosion. *Computer Animation and Virtual Worlds*, 17(2):99–108, 2006.

Y. Chen, L. Xia, T-T. Wong, X. Tong, H. Bao, B. Guo, and H-Y. Shum. Visual simulation of weathering by γ-ton tracing. In *Proceedings of the 32nd Annual Conference on Computer Graphics and Interactive Techniques*, Special Interest Group on Graphics and Interactive Techniques, pages 1127–1133, New York, NY, USA, 2005. ACM Press.

J. Dorsey, A. Edelman, H. Jensen, J. Legakis, and H. Pedersen. Modeling and rendering of weathered stone. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, Special Interest Group on Graphics and Interactive Techniques, pages 225–234, New York, NY, USA, 1999. ACM Press. doi: http://doi.acm.org/10.1145/311535.311560.

A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.

T. Ito, T. Fujimoto, K. Moraoka, and N. Chiba. Modeling rocky scenery taking into account joints. In *Proceedings of Computer Graphics International*, pages 244–247. IEEE Computer Society, 2003.

M. D. Jones, M. Farley, J. Butler, and M. Beardall. Directable weathering of concave rock using curvature estimation. *IEEE Transactions on Vizualization and Computer Graphics*, 16(1), 2010.

B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Co., New York, New York, 1982.

M. Milligan. Geology of Goblin Valley State Park, Utah. In *Geology of Utah's Parks and Monuments*, pages 421–432. Utah Geological Association and Bryce Canyon Natural History Association, 2003.

A. Munshi, ed. The OpenCL Specification, version 1.0, 2009.

F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, Special Interest Group on Graphics and Interactive Techniques, pages 41–50, New York, NY, USA, 1989. ACM Press. ISBN 0-201-50434-0. doi: http://doi.acm.org/10.1145/74333.74337.

P. Pimienta, W. Carter, and E. Garboczi. Cellular automaton algorithm for surface mass transport due to curvature gradients: simulation of sintering. *Computational Materials Science*, 1(1):63–77, 1992.

Tamás Vicsek. Pattern formation in diffusion-limited aggregation. *Physical Review Letters*, 53(24):2281–2284, Dec 1984. doi: 10.1103/PhysRevLett.53.2281.

O. Šťáva, B. Beneš, M. Brisbin, and J. Křivánek. Interactive terrain modeling using hydraulic erosion. In *Eurographics/ACM Symposium on Computer Animation*, pages 201–210. The Eurographics Association, 2008.