3D Sandpiles and Efficient Computation of the Circular Sandpile

---

A Thesis
Presented to
The Established Interdisciplinary Committee for Computer Science and Mathematics
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Alexander H. W. Grant

May 2018

Approved for the Committee
(Computer Science and Mathematics)

_____

David Perkinson

Acknowledgments

First I would like to thank my advisor, Dave Perkinson, for teaching me about sandpiles and guiding me through each step of this journey. Without him I would not have a thesis. I'd also like to thank Cameron Fish, for writing the first version of WebGL Sandpiles that I built off of and for walking me through the codebase and helping me out with a few bugs. Next, I'd like to thank my dad, Steve Grant, for helping me edit. Finally, I'd like to thank my partner, Francisca Garfia, for helping me through this hard, stressful, and terrifying time in my life.

Table of Contents

List of Figures

Abstract

I expand on an Abelian Sandpile simulator created by Cameron Fish for his thesis in 2017 [Fish17]. I added 3D rendering of the sandpile and faster computation of the circular sandpile. In this thesis I go over the methods used to render the sandpile in 3D and the methods for creating the algorithm for the circular sandpile. I also go over some background needed to understand Abelian Sandpiles and 3D Computer Graphics.

Introduction

For my thesis, I developed a simulation of the Abelian Sandpile in JavaScript and WebGL that I refer to as the "sandpiles program." For the moment, the sandpiles program can be accessed at:

`https://people.reed.edu/~davidp/grant/`

In Chapter 1, I present the concept of an Abelian sandpile. I define the identity for a sandpile and the necessary properties of the sandpile that allow the subsequent calculation of the identity. In Chapter 2, I review how graphics cards render 3D images. In Chapter 3, I describe the algorithm I developed to compute the identity on the circular sandpile, the function used in the sandpiles program to compute the identity, and then report how the algorithm performs. In Chapter 4, I detail how I solved some of my biggest challenges in rendering the sandpile in 3D. I then present some of the capabilities of the program.

Chapter 1

Sandpile Theory

1.1 Basic Setup

An Abelian Sandpile is a mathematical model that vaguely resembles how a pile of sand would act. If you add more sand to the center of the pile, at some point the sand in the middle of the pile disperses out to the edges. In the real world physical laws involving gravity and friction govern how sand moves. Likewise, the Abelian Sandpile model has rules that govern the sand's behavior that are meant to model dispersion. These rules are not meant to accurately depict a physical sandpile, but to provide a simple system in which energy builds and then is released after reaching discrete thresholds. Below are the rules that cause this behavior. There are variations on these rules, but these are the ones we will be most concerned with:

1. Grains of sand are placed on points in a grid, with each point connected to four other points or the edge of the pile.

2. If a point has four or more grains of sand on it, it shoots one to each of its neighbors, or off the edge.

We call the second rule the *legal firing rule*. Let's show this with an example. In

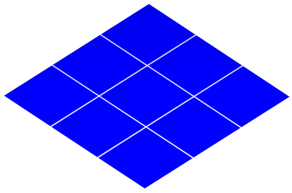

Figure 1.1: An empty sandpile.

Figure 1.1, you see an empty 3×3 sandpile. Let's add some sand. Figure 1.2 shows what it looks like when you add one grain of sand to the center. Figure 1.3 shows three grains of sand on top of each other. This is the highest you can stack the grains without toppling. Figure 1.3 also shows the four colors of the stable heights: zero is blue, one is yellow, two is cyan and three is brown. This is how we color them in the

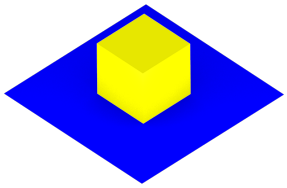

Figure 1.2: One grain of sand.

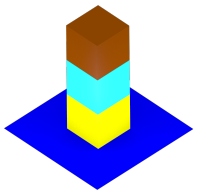

Figure 1.3: Three grains of sand in the center.

sandpiles program. The left sandpile in Figure 1.4 shows one more grain of sand in the center. Unlike in Figure 1.3, this configuration is unstable. Once we apply the firing rule, the sandpile will change. On the right of Figure 1.4 you see the sandpile after the firing rule is applied. The center vertex shoots a grain of sand to its four adjacent vertices. We call the second sandpile *stable* because applying the legal firing rule again would result in the same sandpile. The sandpiles in Figures 1.2 and 1.3 are also stable. To *stabilize* a sandpile is to apply the legal firing rule until you get a stable sandpile. We denote the *stabilization* of a sandpile c as c°.

1.1.1 Sandpiles on Graphs

I've been referring to places on the plane as "vertices" so it's only appropriate that we formally define the graph that a sandpile might live on.

Definition 1. *Let $G = (V, E)$ be an undirected connected graph with vertices V and edges E. We select some $s \in V$ to be the* **sink vertex**. *With this, we can refer to G as a* **sandpile graph**. *We call the number of edges connected to a vertex $v \in V$ the* **degree** *of v, written as* $\deg(v)$.

Example. The $n \times n$ **sandpile grid graph** is an $n \times n$ array of vertices, each vertex connected by edges to its horizontal and vertical neighbors. Vertices on the periphery

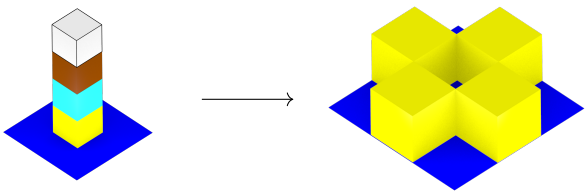

Figure 1.4: Four grains of sand in the center before and after firing rule.

of the array are connected with 1 or 2 edges to the sink vertex, which lives outside the array. All vertices have degree 4, so if a vertex is only connected to 2 normal vertices, it has two connections to the sink.

Figure 1.5 shows the sandpile grid graph for the case $n = 3$. This is the same

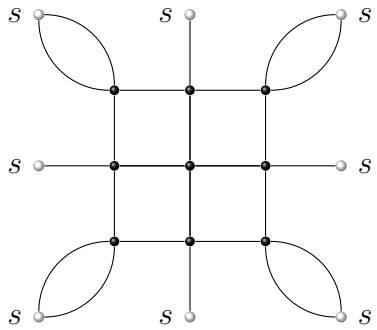

Figure 1.5: Sandpile grid graph. The white balls all represent one vertex, that being the sink.

graph as our 3×3 sandpile. You should note that multiple edges between two vertices are allowed in a sandpile graph, and in the sandpile grid graph, each corner vertex has two edges with the sink. The graph just represents the structures of the plane the sandpile lives on. The sandpile graph is the same across Figures 1.1, 1.2, 1.3, and 1.4. We now need a way to describe the sand on top of the plane.

Definition 2. *Let $G = (V, E)$ be sandpile graph with sink s. Define $\tilde{V} = V \setminus s$. A* **configuration** *on G is an assignment of an integer to every $v \in \tilde{V}$. Ordering the $n := |\tilde{V}|$ vertices of $\tilde{V}$, we identify configurations with elements of $\mathbb{Z}^n$. A* **sandpile** *on G is a nonnegative configuration, i.e., an element of $\mathbb{Z}^n_{\geq 0}$. For $v \in \tilde{V}$ we denote the coordinate of v within a configuration c as $c(v)$.*

Note that the configuration is only defined over $\tilde{V}$, so sand cannot build up on the sink. Earlier I defined the *legal* firing rule. This is an extension of the normal firing rule. Now we have the tools to define the firing rule.

Definition 3. *Given a configuration c on a sandpile graph $G = (V, E)$ and a vertex $v \in V$ we can* **fire** *v which causes v to give one grain of sand to each of its neighbors—including, possibly, the sink—producing a new configuration $\tilde{c}$. In detail, for each $w \in \tilde{V}$,*

$$\tilde{c}(w) = \begin{cases} c(w) + |\{e \in E : e \text{ connects } v \text{ and } w\}| & \text{if } w \neq v \\ c(w) - \deg(v) & \text{if } w = v. \end{cases}$$

It is **legal** *to apply the firing rule at v if $c(v) \geq \deg(v)$.*

Now I also mentioned in the last section what it means to have a stable sandpile. Here is a formal definition for that:

Definition 4. *We call a configuration c* **stable** *if for all $v \in \tilde{V}$, $c(v) < \deg(v)$.*

It also turns out that each stabilization is unique.

Theorem 1. *Let c be a configuration. By repeatedly applying legal firing rules, one arrives at a unique stable configuration, independent of the ordering of the vertices toppled. We call this configuration the* **stabilization** *of c and denote it c°* [CP18, Cor. 6.8, p. 100].

1.1.2 Sandpile Addition

If we have two sandpiles a and b and we write $a + b$, that means that we are placing one sandpile on top of the other.

Definition 5. *For two configurations a, b on sandpile graph $G = (V, E)$ we define $c = a + b$ by $c(v) = a(v) + b(v)$ for all $v \in \tilde{V}$.*

Figure 1.6 shows how this works.

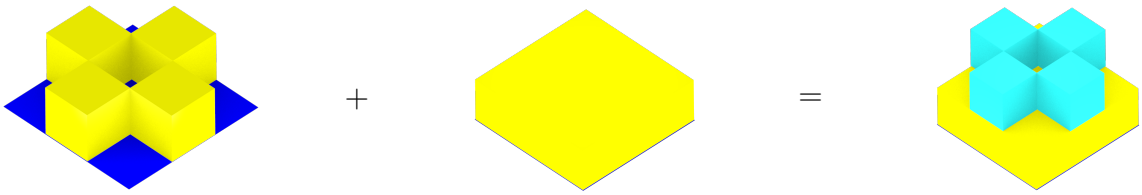

Figure 1.6: Sandpile addition.

1.2 Recurrent Sandpiles

Definition 6. *Let c be a stable sandpile. We call c* **recurrent** *if for any sandpile b, there exists some sandpile a such that $c = (a + b)^\circ$.*

Let's look at some examples. Figure 1.7 shows a recurrent sandpile. How do we

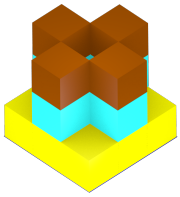

Figure 1.7: A Recurrent sandpile.

know it's recurrent? We can take any stable sandpile, add enough sand so that there are three grains in each cell, add one more to the center and then stabilize. This results in our original sandpile. Figure 1.8 shows this process. The second step shows the maximal stable sandpile, $c_{\max}$. Here is its definition:

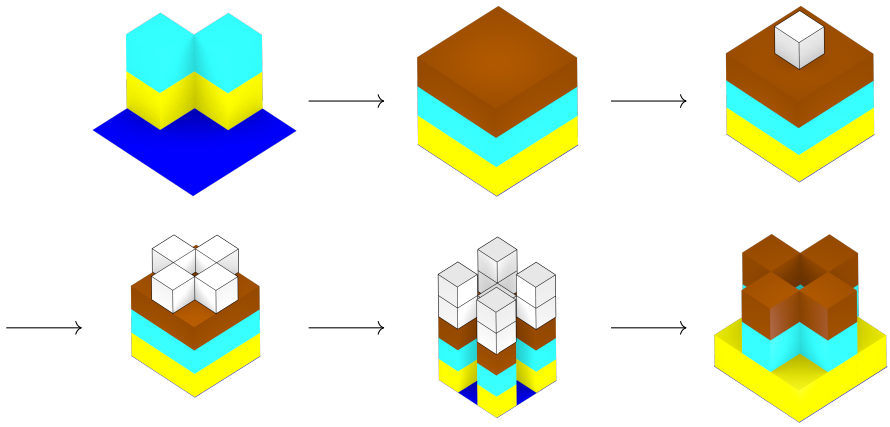

Figure 1.8: An example of how to get to the recurrent sandpile through only adding sand.

Definition 7. *For a sandpile graph $G = (V, E)$, we define $c_{\max}$ by, for all $v \in \tilde{V}$,*

$$c_{\max}(v) = (\deg(v) - 1).$$

The diagram shows one potential starting point to get to our recurrent sandpile. For any stable sandpile, we can easily add sand to get $c_{\max}$. Then after you get there, every other step is the same. So, now we know that it is recurrent. Let's look at an example of a non-recurrent sandpile. Figure 1.9 shows one such sandpile. How do we

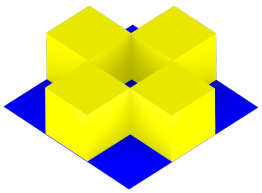

Figure 1.9: A Non-recurrent sandpile.

know it is not recurrent? This theorem will help:

Theorem 2. *A sandpile c on graph G is recurrent if and only if after firing the sink and stabilizing, every vertex of G fires* [CP18, Thm 7.5, p. 120].

Firing the sink means applying the firing rule to the sink. This sends one grain of sand along every edge incident to the sink. For a sandpile grid graph, this ends up adding one grain of sand to every vertex on the edge of the sandpile and two to the vertices in the corners. Firing the sink, we get the stable configuration shown in Figure 1.10, and as you can see, no vertex fires after the sink does because none of the cells reach height 4 or above. By Theorem 2, it cannot be recurrent. We were able to get back to the sandpile in Figure 1.7 because we added a grain sand on top $c_{\max}$ and stabilized. It turns out if we add any amount of sand to $c_{\max}$ and stabilize, we get a recurrent sandpile. In fact,

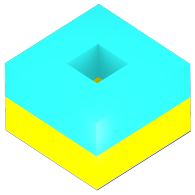

Figure 1.10: The non-recurrent sandpile from Figure 1.9 after firing the sink once.

Proposition 1. *A sandpile a is recurrent iff $a = (b + c_{\max})^\circ$ for some sandpile b.*

1.3 The Laplacian

Let's say we wanted to represent our 3×3 sandpile with a matrix. One way to do that would be to make a row and column for each vertex and if a vertex is connected to another, you put a -1 in the cell of the matrix that lines up to each vertex's row and column. Let's start by making the column for the top left vertex of our 3×3. My labeling system for the matrix can be seen in Figure 1.11.

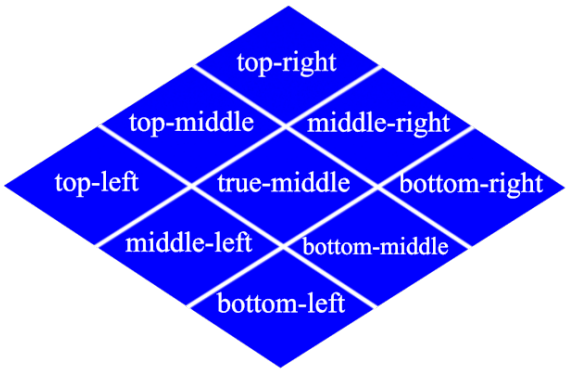

Figure 1.11: Labels for the vertices in the matrix.

$$\begin{array}{c} \\ \text{top-left} \\ \text{top-middle} \\ \text{top-right} \\ \text{middle-left} \\ \text{true-middle} \\ \text{middle-right} \\ \text{bottom-left} \\ \text{bottom-middle} \\ \text{bottom-right} \end{array} \begin{array}{c} \text{top-left} \\ \left[\begin{array}{r} 4 \\ -1 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array}\right. \end{array} \Bigg].$$

So this tells us that the top left vertex is connected to the top middle and middle left vertices. We also put 4 in the cell that lines up to top-left's row and column. This

tells us the degree of top-left, and in a sandpile grid graph, every vertex has degree four. So we know this vertex has two edges to the sink vertex. Let's fill out the rest of the matrix. The columns of the matrix are in the same order left-to-right as its rows are top-to-bottom. So, the leftmost column corresponds to the top-left vertex, the next one to the top-middle, etc.:

$$\begin{array}{r} \text{top-left} \\ \text{top-middle} \\ \text{top-right} \\ \text{middle-left} \\ \text{true-middle} \\ \text{middle-right} \\ \text{bottom-left} \\ \text{bottom-middle} \\ \text{bottom-right} \end{array} \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix}.$$

The above matrix is called the *reduced Laplacian* for the 3×3 grid graph and we denote it $\tilde{L}$. Why is it called "reduced"? That's because we are actually missing a vertex: the sink vertex. If we add that in, we get the usual discrete Laplacian L.

Definition 8. *Order the vertices of a sandpile graph $G = (V, E)$ as $v_1, v_2, ..., v_n$ where $v_n = s$, the sink vertex. The* **Laplacian** *for G is the $n \times n$ matrix*

$$L = D - A$$

where D is the diagonal matrix,

$$D = \text{diag}(\deg(v_1),\ \deg(v_2),\ldots,\deg(v_n))$$

and A is the adjacency matrix for G,

$$A_{ij} = |\{e \in E : e \text{ connects } v_i \text{ and } v_j\}|.$$

The **reduced Laplacian** $\tilde{L}$ *is the $(n-1) \times (n-1)$ matrix that you get by dropping the n-th row and column of L.*

Here is the full discrete Laplacian for our 3×3 sandpile grid graph:

$$\begin{array}{r} \text{top-left} \\ \text{top-middle} \\ \text{top-right} \\ \text{middle-left} \\ \text{true-middle} \\ \text{middle-right} \\ \text{bottom-left} \\ \text{bottom-middle} \\ \text{bottom-right} \\ \text{sink} \end{array} \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -2 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & -1 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 & -2 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 & -1 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 & -2 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -2 \\ -2 & -1 & -2 & -1 & 0 & -1 & -2 & -1 & -2 & 12 \end{bmatrix}.$$

This has the nice property that the sum of all the elements in a column is zero. Dropping the row and column for the sink yields the reduced Laplacian that we saw before.

1.3.1 Using the Laplacian to Fire Vertices on a Sandpile

The Laplacian encodes the firing rule. What I mean by that is that we can use it to apply the firing rule on a vertex:

Proposition 2. *Let G be a sandpile graph with non-sink vertices $v_1, v_2, ..., v_{n-1}$. Let $\tilde{L}$ be the reduced Laplacian for G. Let c be a configuration on G. Firing vertex $v_i \in \tilde{V}$ from c produces the configuration $\tilde{c}$ such that,*

$$\tilde{c} = c - \tilde{L}e_i$$

where e_i is the i-th standard basis vector.

Definition 9. *Now let $\sigma \in \mathbb{Z}^{n-1}$, and let c be a configuration. Define*

$$\tilde{c} = c - \tilde{L}\sigma.$$

Then $\tilde{c}$ is the configuration obtained by firing each vertex v_i a total of $\sigma(i)$ times. We call σ the **firing vector** *that produces $\tilde{c}$ from c.*

If $\sigma(i)$ is negative, then the firing rule is applied in reverse, meaning that v_i will "borrow" sand from its neighbors, increasing its sand count and decreasing theirs. Let's look at an example on the 3×3 sandpile. Here we'll fire the vector $\sigma = (0,0,0,0,1,0,0,0,0)$ on the configuration $c = (0,0,0,0,4,0,0,0,0)$.

$$\begin{bmatrix}0\\0\\0\\0\\4\\0\\0\\0\\0\end{bmatrix} - \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0\\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0\\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0\\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0\\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0\\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1\\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0\\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1\\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\4\\0\\0\\0\\0\end{bmatrix} - \begin{bmatrix}0\\-1\\0\\-1\\4\\-1\\0\\0\\-1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\1\\0\\1\\0\\0\\1\end{bmatrix}.$$

This calculation mirrors the firing shown in Figure 1.4.

1.3.2 Isomorphism of $S(G)$ with $\mathbb{Z}^n/\mathrm{im}(\tilde{L})$, the Cokernel of the Reduced Laplacian

A fundamental result of sandpile theory is that the set of recurrent sandpiles forms a group under the operation of sandpile addition followed by stable addition.

Definition 10. *We define the* **sandpile group**, *$S(G)$, as the set of all recurrent sandpiles on a graph G under the stable addition operation: if a,b are sandpiles then stable addition is $(a+b)^\circ$.*

As noted in the definition of a configuration, any sandpile on a graph with n vertices can be converted into a vector of $\mathbb{Z}^{n-1}$ (since sand does not accumulate on the sink vertex). You order the vertices the same as with the Laplacian (in my case top-left, top-middle etc.) and write down how many grains of sand there are. For example, the sandpile in Figure 1.7 in vector form would be $(1,3,1,3,0,3,1,3,1)$.

Theorem 3. *Let $G=(V,E)$ be a sandpile graph with $n:=|V|$. The mapping,*

$$\begin{aligned} S(G) &\to \frac{\mathbb{Z}^{n-1}}{\mathrm{im}(\tilde{L})} \\ c &\mapsto c+\mathrm{im}(\tilde{L}) \end{aligned}$$

is an isomorphism of groups [CP18, Thm. 6.28, p. 105].

This means that modulo the firing rules, each configuration is equivalent to a unique recurrent sandpile.

1.4 The Identity

1.4.1 Definition

We defined the group $S(G)$, so one may wonder what the identity is for that group? The identity $S(G)$ basically acts like 0 for real numbers. If you add any number to zero, you get that number back. Similarly, if you add any recurrent sandpile to the identity, you get that sandpile back. The sandpile $(0,0,0,...,0)$ has that property, but it is not recurrent. So we want a sandpile that corresponds to the same element of $\mathbb{Z}^{n-1}/\mathrm{im}(\tilde{L})$ as $(0,0,...,0)$ but is recurrent. We state this formally:

Proposition 3. *The identity of $S(G)$ for a graph G with Laplacian L is the recurrent sandpile that is the same as 0^{n-1} modulo $\mathrm{im}(\tilde{L})$.*

1.4.2 Naive Computation of the Sandpile Identity

Proposition 4. *Let c_{id} be the identity for a sandpile graph G. Then*

$$c_{\mathrm{id}} = ((c_{\max} - (2c_{\max})^\circ) + c_{\max})^\circ.$$

Proof. We can write the sandpile on the right as $(a + c_{\max})$ where $a = ((c_{\max} - (2c_{\max})^\circ)$. So by Proposition 1 it is recurrent. Then note that modulo $\mathrm{im}(\tilde{L})$,

$$((c_{\max} - (2c_{\max})^\circ) + c_{\max})^\circ = ((c_{\max} - (2c_{\max})) + c_{\max}) = 2c_{\max} - 2c_{\max} = 0.$$

□

This method is refered to as the *naive method*.

1.4.3 Finding the Firing Vector for the Identity

Definition 11. *The* **firing vector for the identity** *for sandpile graph G is the vector σ_{id} such that*

$$c_{\text{id}} = c_0 - \tilde{L}\sigma_{\text{id}} = -\tilde{L}\sigma_{\text{id}}$$

where c_0 is the configuration with 0 sand at every vertex.

In order to find σ_{id}, we solve the above equation for σ_{id} to get $\tilde{L}^{-1}(-c_{\text{id}}) = \sigma_{\text{id}}$. For example, we compute the identity of the 3×3 sandpile to be $(2, 1, 2, 1, 0, 1, 2, 1, 2)$ using the naive method. Then we calculate the firing vector,

$$\sigma_{\text{id}} = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix}^{-1} \left(- \begin{bmatrix} 2 \\ 1 \\ 2 \\ 1 \\ 0 \\ 1 \\ 2 \\ 1 \\ 2 \end{bmatrix} \right)$$

$$= \begin{bmatrix} \frac{67}{224} & \frac{11}{112} & \frac{1}{32} & \frac{11}{112} & \frac{1}{16} & \frac{3}{112} & \frac{1}{32} & \frac{3}{112} & \frac{3}{224} \\ \frac{11}{112} & \frac{37}{112} & \frac{11}{112} & \frac{1}{16} & \frac{1}{8} & \frac{1}{16} & \frac{3}{112} & \frac{5}{112} & \frac{3}{112} \\ \frac{1}{32} & \frac{11}{112} & \frac{67}{224} & \frac{3}{112} & \frac{1}{16} & \frac{11}{112} & \frac{3}{224} & \frac{3}{112} & \frac{1}{32} \\ \frac{11}{112} & \frac{1}{16} & \frac{3}{112} & \frac{37}{112} & \frac{1}{8} & \frac{5}{112} & \frac{11}{112} & \frac{1}{16} & \frac{3}{112} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} & \frac{1}{8} & \frac{3}{8} & \frac{1}{8} & \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{3}{112} & \frac{1}{16} & \frac{11}{112} & \frac{5}{112} & \frac{1}{8} & \frac{37}{112} & \frac{3}{112} & \frac{1}{16} & \frac{11}{112} \\ \frac{1}{32} & \frac{3}{112} & \frac{3}{224} & \frac{11}{112} & \frac{1}{16} & \frac{3}{112} & \frac{67}{224} & \frac{11}{112} & \frac{1}{32} \\ \frac{3}{112} & \frac{5}{112} & \frac{3}{112} & \frac{1}{16} & \frac{1}{8} & \frac{1}{16} & \frac{11}{112} & \frac{37}{112} & \frac{11}{112} \\ \frac{3}{224} & \frac{3}{112} & \frac{1}{32} & \frac{3}{112} & \frac{1}{16} & \frac{11}{112} & \frac{1}{32} & \frac{11}{112} & \frac{67}{224} \end{bmatrix} \begin{bmatrix} -2 \\ -1 \\ -2 \\ -1 \\ 0 \\ -1 \\ -2 \\ -1 \\ -2 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}.$$

The fact that c_{id} is constant here is an anomaly. For instance, for the 4×4 grid,

$$c_{\text{id}} = (-2, -3, -3, -2, -3, -4, -4, -3, -3, -4, -4, -3, -2, -3, -3, -2).$$

Chapter 2

Computer Background

2.1 Everything is a Triangle

When using a graphics card, any shape you want to show must be represented by a series of triangles in 3D. These triangles are stored by the computer as 3 points in 3D, such as $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$. Those 3 points form the triangle you can see in Figure 2.1. What this means is that any object you want to display must be made

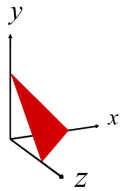

Figure 2.1: Unit triangle in 3D.

out of triangles. Figure 2.2 shows how you would create a cube out of triangles.

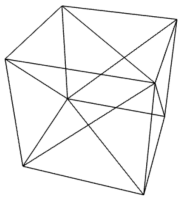

Figure 2.2: Cube made out of triangles.

2.2 The Buffers

A *buffer* is essentially a list of information. In our case, the main RAM and CPU of the computer will hold on to the buffer and the graphics card, which has its own GPU and GRAM will request one piece of information, do some processing on it, then request the next piece. It is more complicated than that since the GPU has hundreds

of processors so it will be working on many bits of information simultaneously. But, for our purposes you can think of a buffer as a list of numbers. The *vertex buffer* will hold the points of the triangles we want to render. If we wanted to make the unit square, our index buffer may look like:

```
[0,0,0,
 1,0,0,
 1,1,0,
 0,1,0]
```

Notice how it is just a list of numbers, not a list of points. The graphics card will read the first three numbers it gets and interpret that as a point, then move on to the next three. The *index buffer* will hold indices into the vertex buffer to define our triangles. To make our square we will need to make two triangles, so our index buffer may look like this:

```
[
// first triangle
 0, // refers to the point (0,0,0)
 1, // refers to the point (1,0,0)
 2, // refers to the point (1,1,0)
// second triangle
 0, // refers to the point (0,0,0)
 2, // refers to the point (1,1,0)
 3, // refers to the point (0,1,0)
 ]
```

Much like how it treats the vertex buffer, the GPU will take the first three indices it sees in the index buffer and interpret that as a triangle before moving on to the next three. Also, notice how the indices do not refer to actual indices in our vertex buffer, but rather what the indices of the points would be if we made a list of 3D points.

2.3 The Graphics Pipeline

The graphics pipeline is the steps your GPU takes to turn triangles defined in three dimensions into pixels on your screen defined in two dimensions. The four stages of the graphics pipeline are vertex processing, rasterization, fragment processing, and blending. The rasterization and blending steps are controlled by the hardware and cannot be changed. The vertex and fragment processing steps can be changed by the programmer.

2.3.1 Vertex Processing

I'm going to describe a very simple vertex shader. The vertex shader is what we call the piece of code that does the vertex processing. First, the CPU sends to the GPU the vertex buffer and the index buffer. The vertex buffer defines the variable

`aVertexPosition` in the code below [Moz15]. The graphics card will take each vertex of each triangle and run the vertex shader on it.

```
1    attribute vec4 aVertexPosition;
2    varying lowp vec4 vColor;
3
4    void main(void) {
5      gl_Position = aVertexPosition;
6      vColor = vec4(1.0, 0.0, 0.0, 1.0) //Red
7    }
```

Whatever point `gl_Position` gets set to is what point gets sent to the rasterer. In this example since we set `gl_Position` to `aVertexPosition`, the point gets sent from the vertex buffer to the rasterer unmodified. We could modify this position if we wanted. For instance we could do

```
gl_Position = aVertexPosition + vec4(0,1,0,0);
```

to make all of the points higher by one. I made a modification like this in the sandpiles program. We create the variable `vColor` here and it gets passed along to the fragment shader.

2.3.2 Rasterization

This step imagines that the triangles are placed behind your computer screen and it checks that if a line can be drawn from your eye to a triangle through a pixel on the computer screen, then it creates a "fragment" at that pixel associated with that triangle. A fragment is a piece of information associated with a pixel on the screen that holds a color, a transparency, and how far the fragment is away from the screen. These fragments get sent to the Fragment Processing step. Figure 2.3 shows how fragments get made. The square on the monitor in the figure represents a fragment.

2.3.3 Fragment Processing

From the rasterization step, we only know the pixel locations of the fragments and how far they are from the screen. In this step we set their colors and their transparencies, otherwise known as alphas. Like the vertex processing step, we call the code that runs this step the *fragment shader*. The code below [Moz15] gets run on every little fragment that the rasterer creates.

```
1    varying lowp vec4 vColor;
2    void main(void) {
3        gl_FragColor = vColor;
4    }
```

So here we set the color of the fragment to the color that we were given by the vertex shader, which is red.

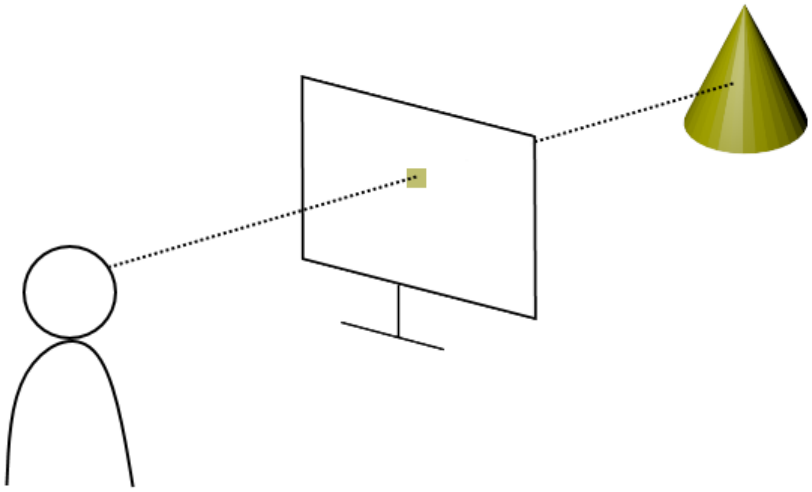

Figure 2.3: The computer decides where to place a fragment on the screen based on the location of the object. The yellow square represents a fragment.

2.3.4 Blending

This part of the graphics pipeline is not important for the sandpiles program so you can skip this section if you want. The `gl_FragColor` variable takes four values: red, green, blue and alpha. This step is where alpha becomes important. Alpha defines how opaque or transparent a fragment is, with an alpha of zero being completely transparent (like glass) and an alpha of one being completely opaque (like a wall). In the blending step, the GPU finds all of the fragments associated with a pixel and then looks at the one closest to the eye of the user. We have the information about distance of the fragment from the Rasterization step. If the first fragment is opaque, then that pixel gets assigned the color of that fragment. If it is not, then that color of that fragment gets "blended" with that of the fragments behind it. This blended color gets assigned to the pixel. The blender does that for every pixel on the screen.

2.4 Transformation Matrices

We pass our objects to the GPU as points via the vertex buffer. Now what if we want to rotate our object to look at it from a different angle? Remaking the vertex buffer with different points would take a while. What if we could come up with a function that takes each point and outputs a new point in a different location and when you apply this function to every point in our object, the object rotates? Well you can, and it turns out that this function is linear. Since it's linear, we know that we can express our function, let's call it F as $F(\mathbf{x}) = A\mathbf{x}$. In the case of rotation about the x-axis, that A is:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

where θ is how much you want to rotate. There are similar matrices for the y- and z-axes. These matrices are not limited to rotation, there are also matrices for translation and scaling. If we have our rotation matrix M, then for every point p in the sandpile we compute $Mp = d$ and display d instead of p. Going back to my earlier sample vertex shader, with a matrix we would set `gl_Position` like this:

```
gl_Position = myTranformationMatrix * aVertexPosition;
```

Chapter 3

Identity for the Circular Sandpile.

3.1 What is the Circular Sandpile?

So far we've been using a square sandpile graph, also known as the sandpile grid graph. You construct the circular sandpile graph from a square sandpile graph by finding the middle of the sandpile graph, then removing every vertex that is more than a certain distance away from the center. As with the square sandpile, all of the vertices that are connected to fewer than 4 "regular" vertices (i.e., the vertices on the periphery of the graph) are connected to the sink. You end up with a blocky circle like that in Figure 3.1.

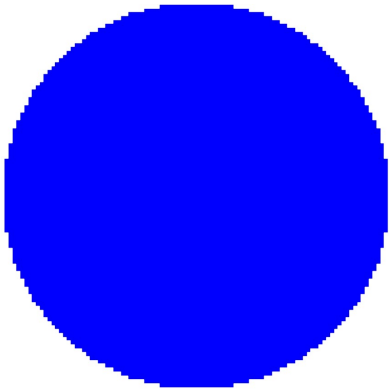

Figure 3.1: Empty circular sandpile.

3.2 Modeling the Firing Vector for the identity

We went over the firing vector for the identity in Chapter 1, but to reiterate: the identity is the recurrent sandpile equivalent to the zero sandpile. We can create the identity with $((c_{\max} - (2c_{\max})^\circ) + c_{\max})^\circ$. We will denote the identity as c_{id}. Recall the firing vector for the identity $\sigma_{\mathrm{id}} = \tilde{L}^{-1}(-c_{\mathrm{id}})$. We'll define the firing script for the identity,

Definition 12. *The* **firing script** *for the identity* c_{id} *is the vector* $s_{\mathrm{id}} = -\sigma_{\mathrm{id}}$.

This means that $c_{\mathrm{id}} = \tilde{L}s_{\mathrm{id}}$ since $c_{\mathrm{id}} = -\tilde{L}\sigma_{\mathrm{id}}$. The firing script tells each vertex how many times it should borrow sand from its neighbors. Unlike the identity which looks mostly random, the firing script looks almost quadratic. Figure 3.2 shows a picture of one firing script. So we tried to model the firing script with a range of

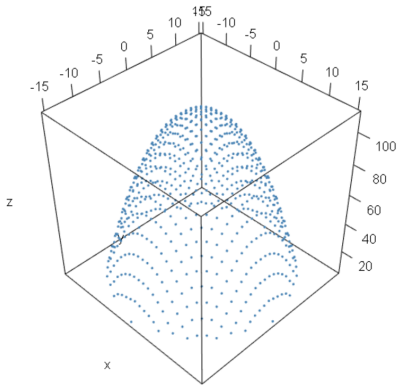

Figure 3.2: Firing script for circular sandpile of diameter 31.

polynomial functions. First we generated a bunch of firing scripts for sandpiles of various diameters using the naive method to get c_{id} and then using $\tilde{L}^{-1}c_{\mathrm{id}}$ to get the firing script. I generated the firing vector for circular sandpiles of diameter 3 to diameter 36. I used least squares regression to find the coefficients for a function of the form $z = b + m(x^2 + y^2)$ to each firing script. I was able to use a function like that and not like $z = b + a_1x + a_2y + a_3x^2 + a_4y^2 + a_5xy$ because the firing script was parabolic and evenly symmetric about the x-axis and y-axis. By doing this I got 34 bs and 34 ms (one for each graph from diameter 3 to 36). I then modeled b as a function of the diameter and the diameter squared and got the function:

$$b = 0.3553 - 0.2145d + 0.1275d^2$$

where d is the diameter. I also modeled m, but the model did not predict m well for large d. You can see the model's graph in Figure 3.3. The model fits the data pretty well, but for a d of 100, it predicts a positive m. All of the firing scripts in the data are concave down, and a positive m would result in a concave up firing script. But, the firing vector is close to zero (around 1 or 2) on the perimeter (i.e., when $x^2 + y^2 = (d/2)^2 = r^2$). So I assume that $z = 0 = b + m(r + 1)^2$ which leads to:

$$m = -\frac{b}{(r+1)^2}$$

which is how I calculate m in the program. Figure 3.4 shows $b + m(x^2 + y^2)$ modeled over our firing script from Figure 3.2.

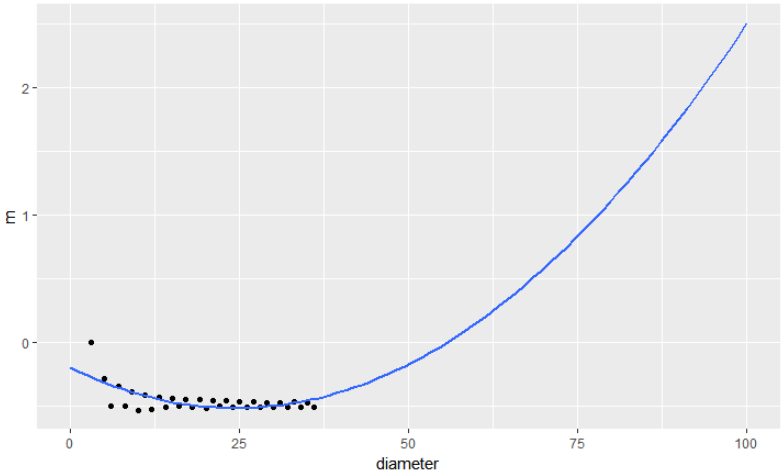

Figure 3.3: Quadratic model of m.

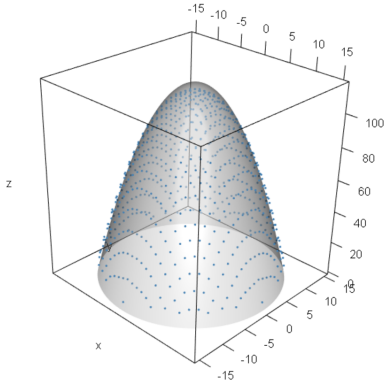

Figure 3.4: Firing script from Figure 3.2 with approximation.

3.3 Confirming the Identity

We've modeled the firing script, but the model is not 100% accurate so we do not know if the model is actually producing the identity. Fortunately, since we produced this configuration, lets call it $c_{?\mathrm{id}}$, with a firing script, we know that it is in the image of the reduced Laplacian (because $c_{?\mathrm{id}} = \tilde{L}s$ for some s). Because it is in $\mathrm{im}(\tilde{L})$, then it is zero modulo $\mathrm{im}(\tilde{L})$. So we have one condition for the identity. Next we need it to be recurrent.

Theorem 4. *Let c be a sandpile. By firing the sink a finite number of times and stabilizing, c is transformed into a recurrent sandpile $\tilde{c}$ such that $c = \tilde{c}$ modulo $\mathrm{im}\tilde{L}$* [CP18, Thm 7.5 p. 120].

Corollary 1. *Let $\tilde{c}$ be the result of firing the sink on a sandpile c. If $c = \tilde{c}$ then c is recurrent.*

Cameron Fish wrote a function that will fire the sink, check if the configuration changes, then continue firing until it does not change. I applied that function to the

configuration resulting from my model for each diameter, 1–510, and then recorded how many times it needs to fire until it does not change. No circular sandpile in the range tested changed after applying my model and firing the sink. By Corollary 1, my algorithm will produce the identity for every circular sandpile of diameter 1–510.

3.4 Results

I tested the naive method against my method on a circular sandpile of diameter 510 and timed it with a stopwatch. On a desktop computer with an Intel Xeon E5-1607 and an NVIDIA Quadro K2000 running Firefox 50 the computation time was 28.48 seconds for the naive method and 1.04 seconds for my method. On a laptop computer with an Intel Core i5-6200U and Intel HD Graphics 520 running Firefox 59 the computation time was 1 minute 39 seconds for the naive method and 1.74 seconds for my method.

Chapter 4

Creating 3D sandpiles

4.1 The Plane

The 3D sandpile in the sandpiles program is made up of a plane of around 10000 points in a grid, although this can be adjusted. The sandpile by default is 100×100. I chose to have the plane go across the x and z axis and have the sand stack on the y axis. I made it so our 10000 points range from 0–100 on the x axis and 0–100 in the z axis, or (0,0,0) to (100,0,100). Each point is surrounded by eight triangles that connect it to the eight surrounding points. You can see an image of this in Figure 4.1. Each of these points represents a certain location on the sandpile. As I said earlier,

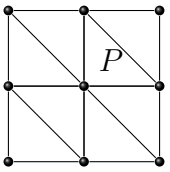

Figure 4.1: Section of 3D sandpile grid.

the y value of the point represents how high the sand is stacked there. Each point is assigned a color depending on its height. However, the color does not show up on the point, it shows up on the triangle. So, the triangle interpolates the color between its three points. One corner of the triangle may be brown, another blue and another yellow. It will smoothly transition between these colors in the middle.

4.2 The Textures

A texture is a 2D image that you map on to a 3D object. This lets you show a lot of detail on 3D models. The graphics shaders are able to control how the texture gets displayed on the model. In our case, this means that we just use the texture as a storage medium for the sandpile between the graphics shaders and the JavaScript. We have two textures called “front” and “back”. We apply the firing rule in a graphics shader. The graphics shader will read from the “front” texture, apply the rule, and

write the sandpile after the application to the "back" texture. We swap these textures when we want to apply the rule next.

When rendering the 3D model we send a square of points to the vertex shader that covers the space. We map each point in the square to the texture and it will tell us if the vertex is in the shape or not. If not, we do not render the vertex. The texture also tells us the height (how much sand is stacked up) at each vertex. We then adjust the y coordinate of the vertex according to the height stored in the texture.

The points I send to the vertex shader are all of the form $(x, 0, z)$. All of the xs and zs are integers between 0 and the grid size of the sandpile (default 100). Let's refer to the grid size as g. The sandpile is in the center of the texture, but the texture has a bunch of empty "sink" vertices surrounding the sandpile. The texture is 512 by 512. As a result of this, the maximum width or height of a sandpile in the program is 510. To line up the x value of a given point, I look at the location in the texture at $x + (512 - g)/2$.

Since our points are from 0–g, this makes them line up nicely with the texture, but if we were to display them, the sandpile would look funny. The view of the 3D model is centered at (0,0,0), so since all of the sandpile points are greater than zero, it would be in the top right of the screen. So, I use a transformation matrix to align them. I will talk about this in the next section.

4.3 Linear Transformations in the Sandpile Program

Most obviously we use a transformation matrix to rotate the sandpile when the user clicks and drags. Dragging the mouse along the x axis of the screen (side to side) rotates it over the 3D x axis, and dragging it in the y axis (up and down) rotates it over the 3D y axis. So, I also need to scale and center the points to get them from the range 0–100 to -0.5–0.5. First I want to center the sandpile at zero, then I want to scale it so that the points are between -0.5 and 0.5. I make the appropriate scaling and translation matrices, let's call them S and T respectively. Then I want to do TSp where p is a point in the sandpile. Calculating TS for every point would be expensive, so I compute TS in JavaScript and set it to another matrix, let's call it G. Then in the vertex shader I compute Gp, which is much faster. My full command for setting `gl_Position` in the vertex shader is:

```
gl_Position = uProjectionMatrix * uModelViewMatrix * uGridMatrix
    * (aVertexPosition + vertexAdder);
```

where `uModelViewMatrix` has my rotation, `uGridMatrix` is my G from above, `aVertexPosition` is the point as passed from JavaScript, `vertexAdder` holds the height I got from the texture. I don't want to explain `uProjectionMatrix` because I computed it with a library.

4.4 Result

All of this produces 3D images of the sandpile like that which you can see in Figure 4.2.

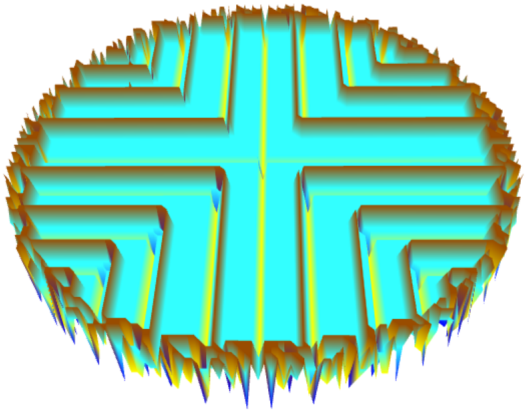

Figure 4.2: 3D rendering of the identity on the circular sandpile.

References

[CP18] Corry, Scott and David Perkinson, *Divisors and Sandpiles*, To appear, Providence, Rhode Island, American Mathematical Society, 2018. (http://people.reed.edu/~davidp/divisors_and_sandpiles).

[Moz15] Mozilla, *webgl-demo.js*, 2016. https://github.com/mdn/webgl-examples/commits/gh-pages/tutorial/sample5/webgl-demo.js.

[Fish17] Fish, Cameron, "A GPU approach to the Abelian sandpile model", Undergraduate Thesis, Reed College, Portland, Oregon, 2017. http://people.reed.edu/~davidp/homepage/students/fish.pdf

[AS15] Angel, Edward and Dave Shreiner, *A Top-Down Approach with WebGL*, 7th Ed., Upper Saddle River, New Jersey, Pearson Education, Inc., 2017.