A GPU approach to the Abelian sandpile model

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

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

Cameron Alexander Fish

May 2017

Approved for the Division
(Mathematics)

David Perkinson

Acknowledgements

I am extremely grateful for the guidance and support of David Perkinson throughout this project. His insights prompted most of mine. I give thanks also to Jim Fix, who believed in me and taught me most of what I know about programming. I am grateful also to Ray Mayer, Joel Franklin, Albyn Jones and John Lind, all of whom showed interest in this project and provided invaluable suggestions and insight. Thank you also to Zhenya Bershtein and Lucas Illing who (along with David and John) heard my oral examination and asked lots of good questions! I would also like to thank Christopher Wellons, whose Game of Life implementation in WebGL provided inspiration and structure for my own WebGL implementation of sandpiles. Thanks also to Wesley Pegden and Lionel Levine for their interest and useful advice. I was also delighted to see some of our images in Luis Garcia and Brady Haran's informational video on sandpiles–thanks for spreading the word!

I am also extremely thankful for the love and support of my family and friends throughout this year and every year. Without them, I could never have stuck with a project like this for so long.

Lastly, thanks to Reed College and to all my professors, each of whom taught me much and gave me the chances I needed.

Table of Contents

List of Figures

Abstract

The Abelian sandpile model provides examples of groups with highly "non-trivial" identity elements. These elements are, at least in the case of sandpile groups on grid graphs, visually stunning. An appreciation of these visuals can be more than an aesthetic one, as they also serve to guide intuition and suggest further routes of study. However, these elements are in general difficult to compute, especially when the underlying graph becomes large. We make use of GPU computation to develop a new framework for the simulation and display of sandpiles, as well as suggest several methods for more efficient calculation of the identity sandpile on grid graphs.

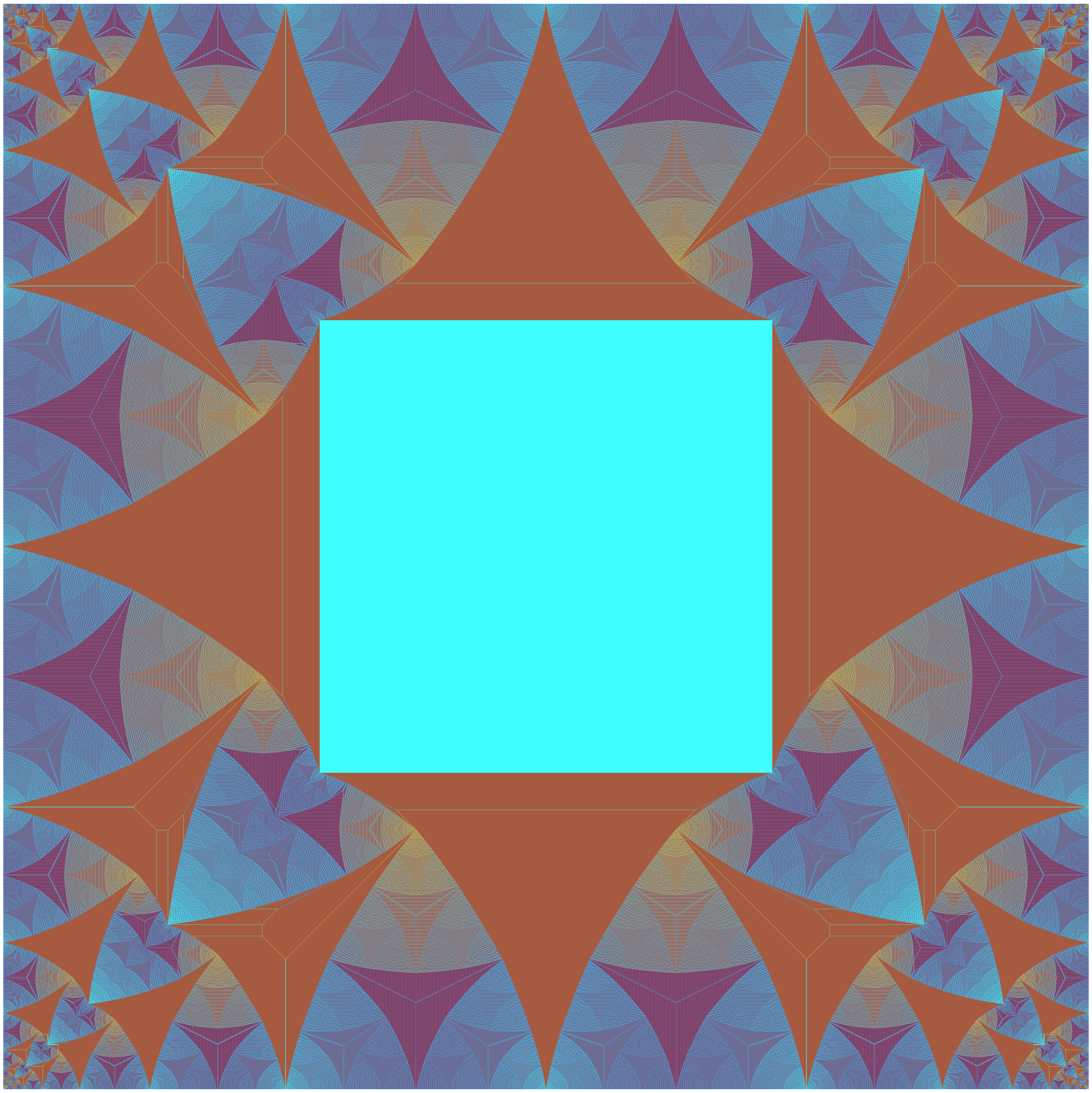

Figure 1: The identity element of the Sandpile group on a 4000×4000 grid graph.

Introduction

Imagine trickling sand onto a tabletop, one grain at a time. A small pile grows. New grains tumble down the sides of the pile, perhaps knocking down others along the way. Eventually, the grains will settle down. Some will come to rest where they are and some will slip off the table entirely. The Abelian sandpile model may be thought of as an attempt to capture some of this behavior, and happily we discover that this simple model produces some impressive visuals and some interesting mathematics, both of which are the subject of this thesis.

To formalize the above image, consider a grid of cells into each of which we may drop any number of grains of sand. Whenever a cell contains four or more grains, it is unstable and will *topple*, dispensing a single grain to each of its four neighbors. Should subsequent cells also contain four or more grains, they too will topple, and so on. We can see that these rules easily allow for a cascade of toppling. Consider a grid with each cell containing three grains of sand. None are unstable, yet the addition of a single grain somewhere on the grid creates an expanding diamond of unstable cells (Figure 2).

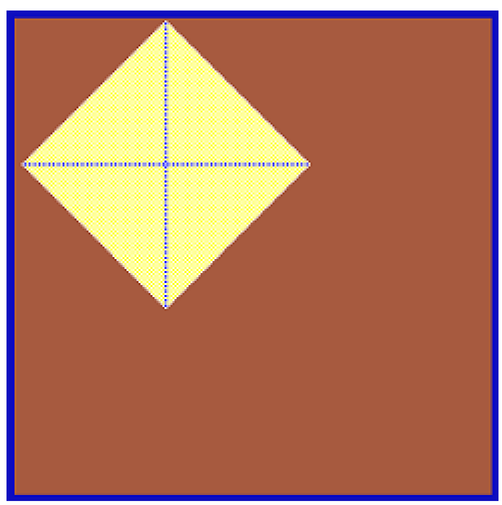

Figure 2: This color scheme will be used throughout—dark blue for 0 grains, yellow for 1 grain, light blue for 2 grains, and brown for 3 grains. Consider how the grain placed (in the epicenter of this diamond) causes its immediate neighbors to become unstable, which then destabilizes their neighbors, and so on.

While it is possible to consider this process on infinite grids, we here restrict ourselves to finite grids, meaning that such a propagation cannot continue forever. To capture the table analogy, we give this grid a boundary where sand falls off. Cells on the boundary of the grid will send grains into the void, removing them from the grid entirely. What happens when this expanding diamond reaches the boundary (Figure 3)?

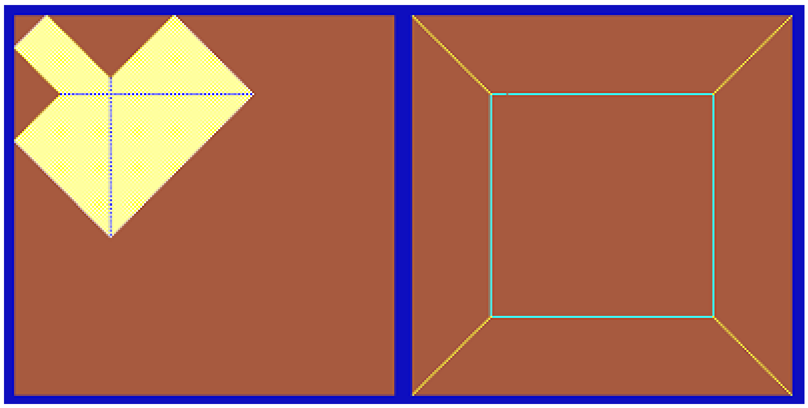

Figure 3: The stabilization of the all 3s sandpile with one grain added. Notice the triangles of brown height 3 cells at the top and left–these are the result of the first "rebound" where the expanding diamond reaches the boundary.

After several "rebounds" like this, every cell has become stable. We call this entire process *stabilization*.

As long as at least one cell is a boundary cell, any initial configuration of sand will stabilize. Without such a boundary, some initial configurations will stabilize and some will not, depending on the number of initial grains. Although the grid passes through numerous states on the way to a stable one, we are primarily concerned with stable configurations, and in particular a subset of the stable configurations which are *recurrent*. We will more carefully define recurrent configurations later, but for now we can say that every stabilization of the kind just illustrated is recurrent (the all 3s configuration with any grains added to any of its cells). It turns out that if we add any two recurrent configurations (each cells grains are added together) and then carry out this stabilization process, the resulting stable configuration is itself recurrent. In fact, these recurrent configurations with this add then stabilize operation actually form a group!

What is the identity of this group? The obvious candidate of the empty configuration is unfortunately not recurrent. We shall see that finding the identity element in general is difficult. The following image perhaps illustrates the complexity of the problem (Figure 4).

The identity element turns out to be strikingly complex. Why is there a square in the middle? Why the fractal appearance? Why these strange lines in the corners? Even more interesting, perhaps, is the consistency with which such features appear as we vary the size of the grid (Figures 5-7). Such features even appear regularly without directly invoking the identity. Consider some of the following images (Figures 8-9).

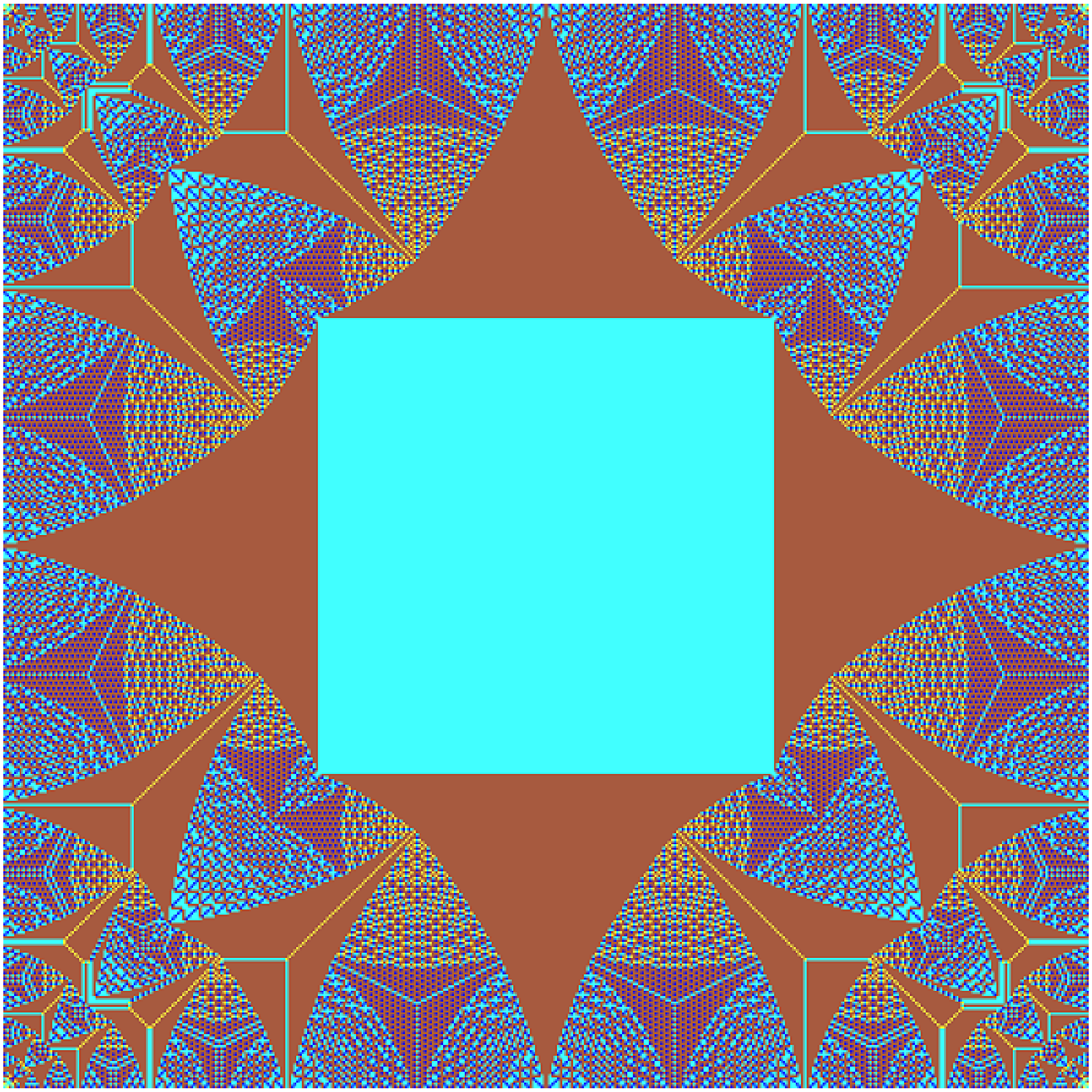

Figure 4: The identity on a 400×400 grid.

It seems plausible that a proper explanation of these features would provide a deeper understanding of the structure and dynamics of the sandpile model as a whole. To that end, it would be very useful to be able to produce identity elements on grids of any size or shape. The identity elements on larger grids in particular have much detail and reveal more of their structure.

However, previous approaches to producing these identities have been computationally intensive. As such, our goal with this project has been to find more efficient methods. We have found significant improvements through highly parallelized GPU computation, and have also developed some empirical methods for quickly computing the identity.

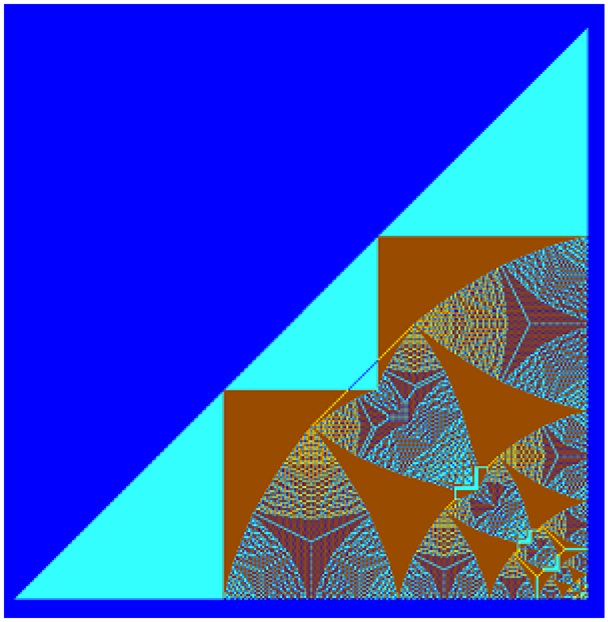

Figure 5: The identity on a triangular grid.

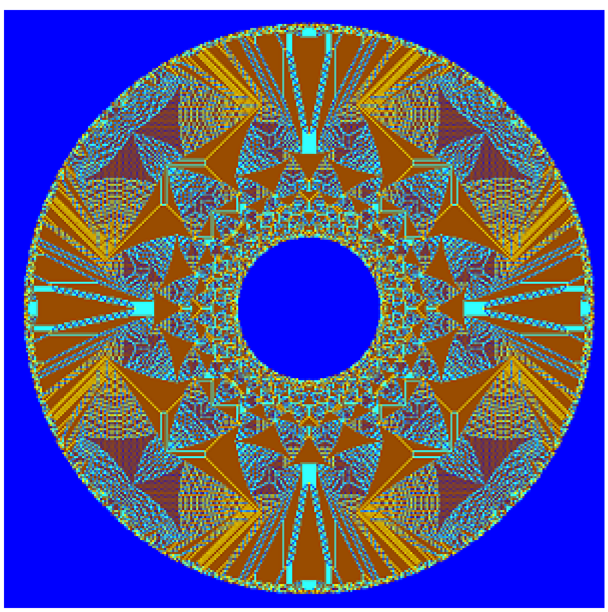

Figure 6: The identity on a ring grid.

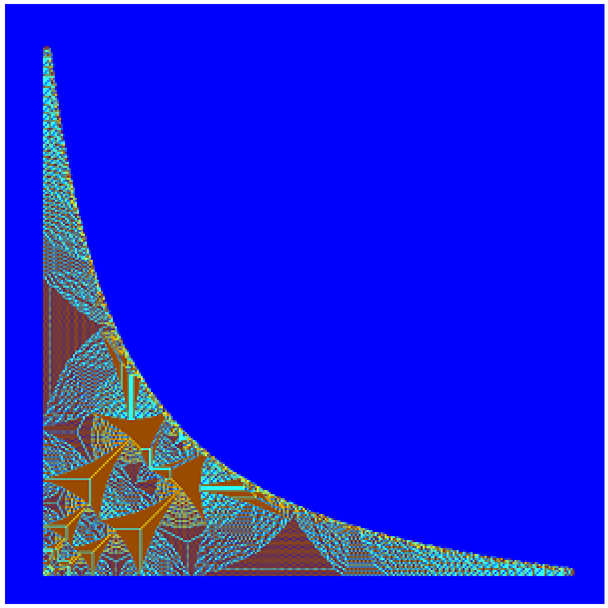

Figure 7: The identity on a "hyperbola" grid.

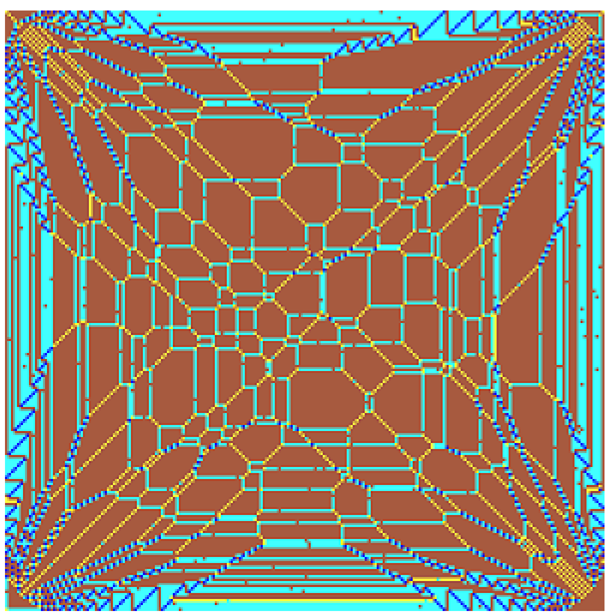

Figure 8: The stabilization of all 3s plus some random grains.

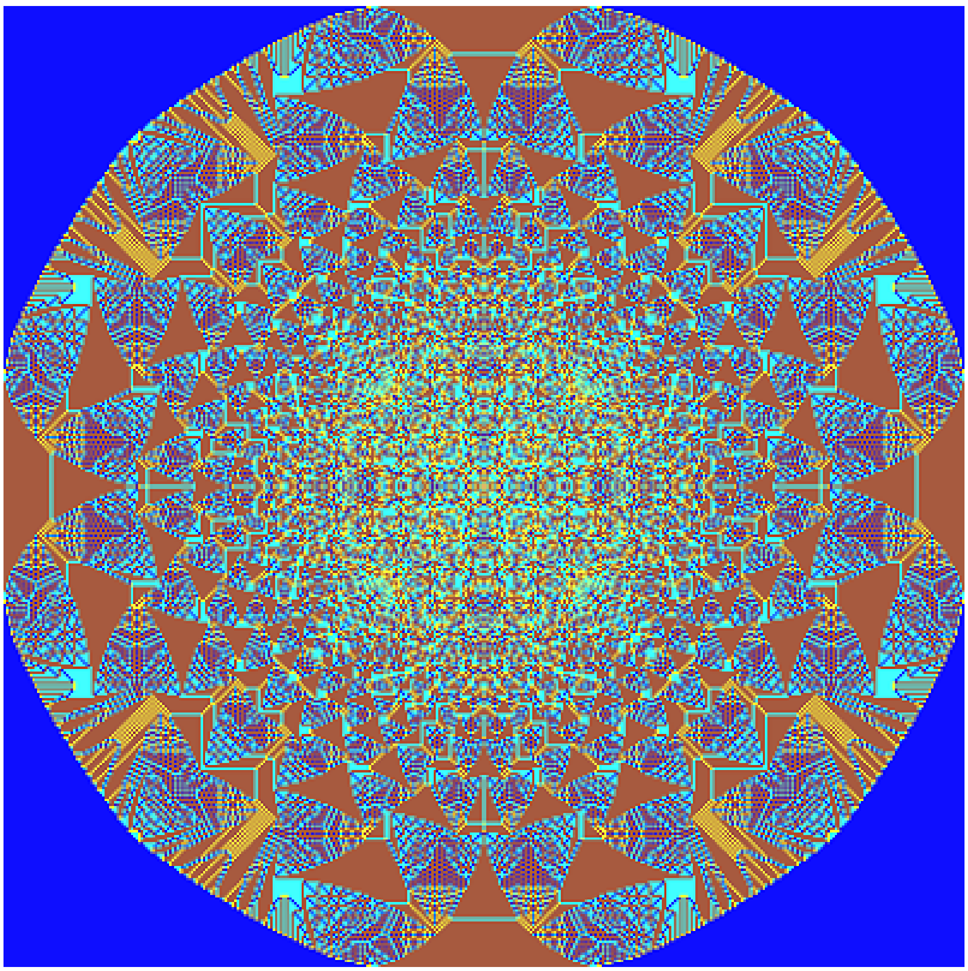

Figure 9: The stabilization of a large number of grains placed in the center.

Chapter 1

Sandpile Groups

Here we shall take the time to more formally define these sandpile configurations. While a lot of interesting mathematics is associated with the theory of sandpiles, we will here focus on the basic definitions and concepts which are necessary to discuss our aims and our results.

1.1 Stabilization

In the above discussion, we referred only to sand grains placed onto a grid. While this scenario is our main focus, sandpiles are typically defined on more general graphs. Consider a connected undirected graph $G = (V, E)$ with vertices $v_1, v_2, \ldots, v_{n+1}$ and edges E. As mentioned above, we would like every configuration to stabilize, so we designate vertex v_{n+1} as the "sink" vertex. We will usually imagine that sand landing on this vertex disappears.

The *degree* $\deg(v)$ of a vertex v is the number of edges connected to v. For the $n \times n$ grid graph, for example, there are n^2 vertices (i, j) with $1 \leq i, j \leq n$ and a sink vertex s with one edge to each border vertex and two edges to every corner. Every non-sink vertex in this graph has degree 4.

A *configuration* on G is an integer vector $c = (c_1, c_2, \ldots, c_n)$ which assigns an integer c_i to each vertex v_i. We will think of these integers as representing the amount of sand present at each node. Such a configuration is a *sandpile* if each $c_i \geq 0$.

If any node contains too much sand, it *fires* (or *topples*), sending some of its own grains to its neighboring nodes. Above, we specified a threshold of *four* grains, but this was for the special case of grid graphs where each node has four neighbors. For graphs in general we let a node topple when it has exactly as many grains of sand as neighbors. This choice of threshold is somewhat arbitrary, but is motivated by a desire for the toppling of a node to send a grain to each one of its neighbors. Below is an example of this firing (Figure 1.1).

To formally capture this firing process, we define the *reduced Laplacian* matrix L for G. Let D be the $n \times n$ diagonal matrix whose ith diagonal entry is $\deg(v_i)$ and let A be the adjacency matrix for G whose (i, j)th entry is the number of edges connecting v_i to v_j. The reduced Laplacian L is then $D - A$. Note that the sink

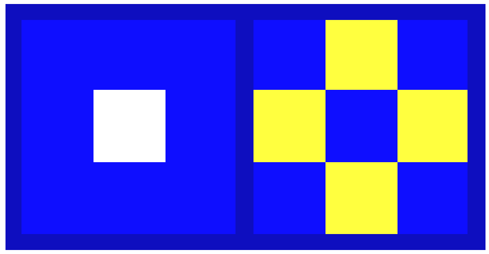

Figure 1.1: The 3×3 grid with 4 grains in the middle, followed by its stabilization.

vertex v_{n+1} is not explicitly part of the construction of L.

Identify v_i with the ith standard basis vector for $\mathbb{Z}^n$. Then if c and c' are configurations where c' is obtained by firing from c by firing some vertex v, we have:

$$c' = c - Lv$$

Thus the result of firing a vertex v_i a total of σ_i times for $i = 1, \ldots, n$ is $c' = c - L\sigma$ where $\sigma = (\sigma_1, \ldots, \sigma_n)$. We call σthe*firing vector*$(or$ *firing script*$) taking c to c$'.$By the matrix-tree theorem, the determinant of$ L $is the number of spanning trees of$ G, $and hence the determinant of$ L $is non-zero. In particular,$ L $is invertible and so the firing vector is unique.$

For example, suppose $c = (0, 4, 0, 0)$ and L is the reduced Laplacian for the 2×2 grid graph (Figure 1.2). Let v be the firing script $v = (0, 1, 0, 0)$ (we are going to fire the second vertex). Then:

$$c' = c - Lv = (0, 4, 0, 0) - (-1, 4, 0, -1) = (1, 0, 0, 1).$$

```
[ 4 -1 -1  0]
[-1  4  0 -1]
[-1  0  4 -1]
[ 0 -1 -1  4]
```

```
[ 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]
```

```
[ 4 -1  0  0 -1  0  0  0  0  0  0  0  0  0  0  0]
[-1  4 -1  0  0 -1  0  0  0  0  0  0  0  0  0  0]
[ 0 -1  4 -1  0  0 -1  0  0  0  0  0  0  0  0  0]
[ 0  0 -1  4  0  0  0 -1  0  0  0  0  0  0  0  0]
[-1  0  0  0  4 -1  0  0 -1  0  0  0  0  0  0  0]
[ 0 -1  0  0 -1  4 -1  0  0 -1  0  0  0  0  0  0]
[ 0  0 -1  0  0 -1  4 -1  0  0 -1  0  0  0  0  0]
[ 0  0  0 -1  0  0 -1  4  0  0  0 -1  0  0  0  0]
[ 0  0  0  0 -1  0  0  0  4 -1  0  0 -1  0  0  0]
[ 0  0  0  0  0 -1  0  0 -1  4 -1  0  0 -1  0  0]
[ 0  0  0  0  0  0 -1  0  0 -1  4 -1  0  0 -1  0]
[ 0  0  0  0  0  0  0 -1  0  0 -1  4  0  0  0 -1]
[ 0  0  0  0  0  0  0  0 -1  0  0  0  4 -1  0  0]
[ 0  0  0  0  0  0  0  0  0 -1  0  0 -1  4 -1  0]
[ 0  0  0  0  0  0  0  0  0  0 -1  0  0 -1  4 -1]
[ 0  0  0  0  0  0  0  0  0  0  0 -1  0  0 -1  4]
```

Figure 1.2: The reduced Laplacian for the 2×2, 3×3, and 4×4 grid graphs.

We can use the reduced Laplacian to describe stabilization in the following way. A vertex v_i in the configuration c is stable if $c_i < \deg(v_i)$. We say c as a whole is *stable* if each (non-sink) vertex is stable. Since every vertex is connected by a sequence of edges to the sink, every configuration can be stabilized by firing a sequence of unstable vertices (note c can be stable regardless of the amount of sand on the sink). We denote the stabilization of c by $\mathrm{stab}(c)$. It is a well-known result that the stabilization is unique (and independent of the order of the vertex-firings).

While c and $\mathrm{stab}(c)$ may be different configurations of sand, we would like to be able to say they are equivalent in the sense that c "collapses" into $\mathrm{stab}(c)$ simply by firing unstable vertices until it is stable. Note that $c - \mathrm{stab}(c) = c - c + Lv = Lv$, that is that they differ only in that some vertices have been fired, as opposed to completely new grains of sand being added, for example. Thus we can say two configurations are *linearly equivalent* if they are equivalent modulo the image of the reduced Laplacian, as $\mathrm{im}(L)$ is the set of all possible ways a configuration may change after some cells have been fired. More simply, c and c are *linearly equivalent* when there exists some v such that $c = c - Lv$.

1.2 Recurrents

On any of these graphs, it is clear that there are an enormous number of stable configurations. For example, on a 10×10 grid, every cell in a stable configuration can have 0, 1, 2, or 3 grains, so there are $4^{100} \approx 1.6 \cdot 10^{60}$ stable configurations. In general, the number of stable configurations is $\prod_{v_i} \deg(v_i)$, a staggering number for all but the smallest graphs. However, many of these stable configurations seem little more than noise (Figure 1.3).

If we imagined dropping a number of grains into random cells, it seems vanishingly likely that any particular one of these noisy configurations would be reached. One may wonder if any particular configurations are likely to be reached at all. We can test this theory explicitly (Figure 1.4). It turns out that there is indeed a set of stable configurations which are seen much more commonly than others during this experiment. Moreover, once one configuration in this set is reached, all further configurations are also in this set (the set is closed under adding a random grain and stabilizing). We call this set of stable configurations the *recurrent* configurations. These configurations appear with probability approaching 1 as the number of grains dropped approaches infinity. Figure 1.4 shows the result of 10 trials of an experiment in which 100 grains of sand are randomly dropped on vertices of the diamond graph. After a grain is dropped, the sandpile is stabilized. The table records how many times each stable configuration is reached. It turns out there are eight recurrent sandpiles on this graph, consistent with the results of this experiment[1]

[1] For more details, see Perkinson (2016).

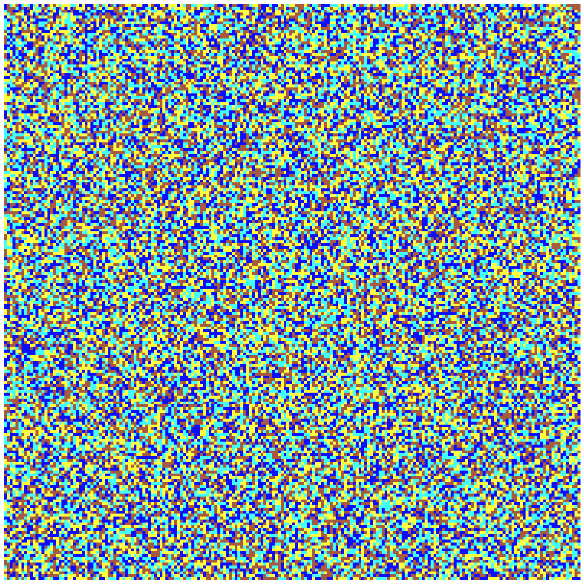

Figure 1.3: A random stable configuration

Sandpile	Trials									
(0, 0, 0)	0	0	0	0	0	0	0	0	0	0
(0, 0, 1)	0	0	0	1	0	1	0	1	1	0
(0, 1, 0)	0	0	1	0	0	0	1	0	0	1
(0, 1, 1)	0	0	0	0	0	0	0	0	0	1
(0, 2, 0)	0	0	1	0	0	0	1	0	0	0
(0, 2, 1)	11	8	11	11	16	14	13	12	9	16
(1, 0, 0)	1	1	0	0	1	0	0	0	0	0
(1, 0, 1)	0	1	0	0	1	0	0	1	0	0
(1, 1, 0)	0	0	0	1	0	1	0	0	1	0
(1, 1, 1)	0	0	0	0	1	0	0	0	0	0
(1, 2, 0)	12	14	13	15	9	11	10	12	18	15
(1, 2, 1)	16	14	16	12	13	7	13	12	12	13
(2, 0, 0)	1	0	0	0	0	0	0	0	0	0
(2, 0, 1)	15	11	9	16	8	17	7	10	10	15
(2, 1, 0)	7	12	15	13	11	16	16	17	9	11
(2, 1, 1)	17	15	13	10	7	15	14	15	6	8
(2, 2, 0)	6	11	9	12	16	12	12	10	21	10
(2, 2, 1)	14	13	12	9	17	6	13	10	13	10

Figure 1.4: Frequency across 10 trials of sandpile occurence when dropping 100 random grains.

We now define these recurrent configurations explicitly. A configuration c on a graph is *recurrent* if:

- $c \geq 0$
- c is stable
- For every configuration a, there exists a configuration $b \geq 0$ such that $c = stab(a + b)$.

We mentioned previously that the stabilization of the all 3's configuration plus any other configuration is recurrent. With this definition, we can see that the maximal stable configuration $c_{\max}$ (all 3's in the grid graph case) is recurrent. The first two conditions are clear, and for the third consider that for any stable configuration a, there exists a configuration $b \geq 0$ such that $a + b = c_{\max}$. So for all a, there exists a b such that $\text{stab}(a + b) = \text{stab}(a) + \text{stab}(b) = c_{\max}$. It follows that any configuration c is recurrent if there is a configuration $b \geq 0$ such that $c = \text{stab}(c_{\max} + b)$.

Let $S(G)$ denote the set of recurrents on G. It turns out these recurrent configurations form a group (called the *Sandpile group* on a graph), under the operation $a \oplus b := \text{stab}(a + b)$. It is well-known that each configuration is linearly equivalent to some unique recurrent, thus giving the group isomorphism:

$$S(G) \approx \mathbb{Z}^n / \text{im}(L).$$

As we have seen (Figure 4), the identity of the Sandpile group $S(G)$ is non-trivial. However, since the equivalence class of 0 in $\mathbb{Z}^n/\text{im}(L)$ is the identity and group homomorphisms preserve the identity, we do know that $id = L\sigma_{id}$ for a unique firing script σ_{id}. This means that the identity is the unique configuration which is both recurrent and linearly equivalent to zero. So one way to find the identity is to compute:

$$\text{stab}((c_{\max} - \text{stab}(2 \cdot c_{\max})) + c_{\max})$$

Another straightforward method involves a special configuration called the *burning configuration*, defined as the the configuration $b = L1$ where 1 is the all-ones vector. This is the configuration obtained by starting with the all-zeroes configuration and firing the sink (Figure 1.6). Note that any multiple of b is linearly equivalent to 0. Consider the stabilization of kb for some large integer k. By selectively firing vertices, we can obtain a configuration which is $c_{\max} + a$ for some a. We know the stabilization of this configuration is recurrent. Hence $\text{stab}(kb) = id$ for large k.

We can use this fact to compute the identity on a grid graph. Simply fire the sink and stabilize repeatedly until the configuration does not change further.

These methods allow us to calculate the identity on any graph. However, actually carrying out these calculations by hand is implausible for all but the smallest of graphs. For this reason we turn to computation.

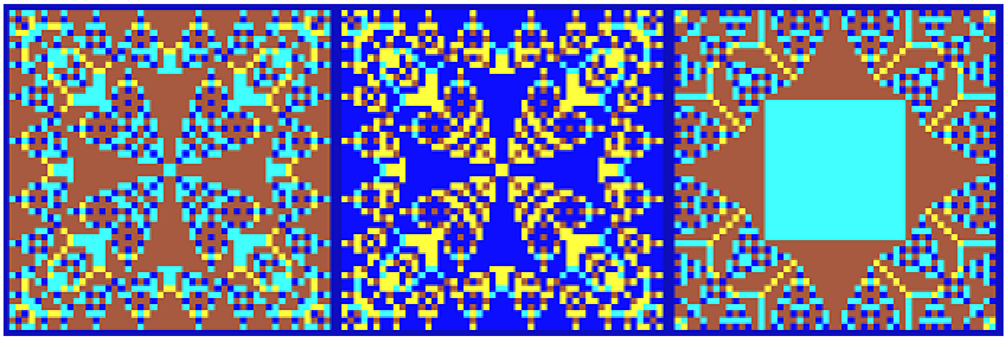

Figure 1.5: The stabilization of $2 \cdot c_{\max}$, then $c_{\max} - \text{stab}(2 \cdot c_{\max})$, then $\text{stab}((c_{\max} - \text{stab}(2 \cdot c_{\max})) + c_{\max})$.

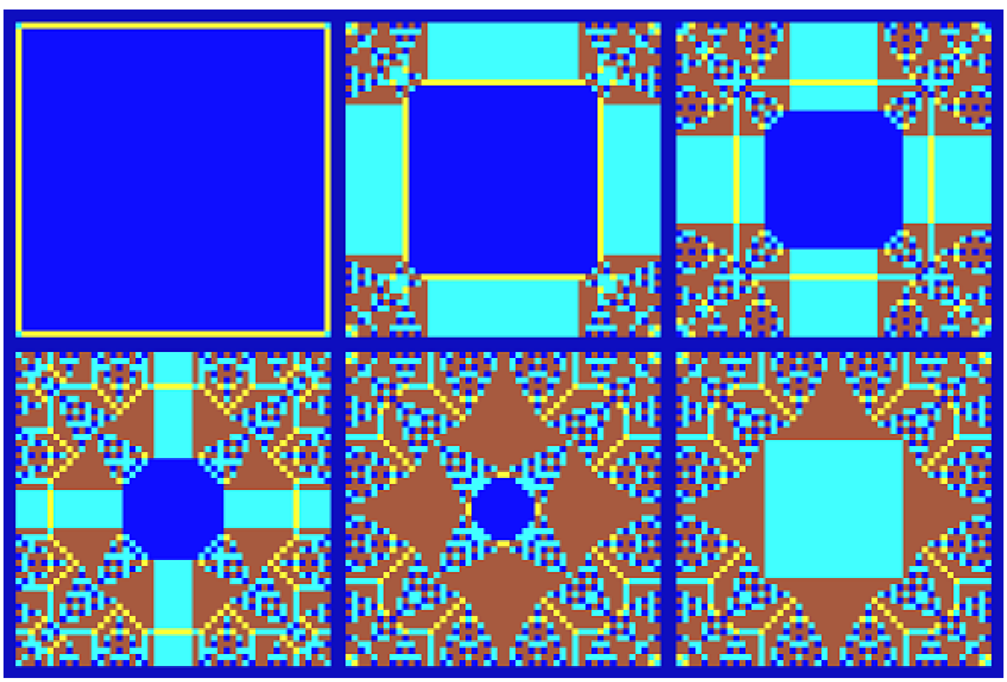

Figure 1.6: The stabilization of kb on the 100×100 grid graph for $k = 1$, $k = 100$, $k = 200$, $k = 300$, $k = 400$, and $k = 500$.

Chapter 2

GPU Computation

Storing grid configurations and adding them together is as straightforward as storing and adding arrays. The difficulty comes in carrying out the stabilization process. One approach is to loop through each cell to check which are unstable, fire each (subtract 4 and give 1 to each neighbor), then repeat until no unstable cells are found. As discussed previously, the firing order doesn't matter, so this method could be implemented in a number of ways which all work. One could fire all unstable cells at once for example (thinking of this as one "frame" of an animation of the firing process), or fire all the unstable cells in one region first, or fire the first unstable cell found, etc. These approaches all suffer from unnecessary looping. It is difficult to know what effect a single firing will have on the sandpile as a whole, so finding some optimal firing order (to minimize the number of loops) is impractical, and possibly even more difficult than simply carrying out the computation.

One useful insight is that when considering a single "frame" of stabilization (that is, the simultaneous firing of each unstable cell), every firing can at most affect only 5 cells (the firing cell itself and its four neighbors). This means that on a frame-by-frame basis, each cell only needs information about itself and its neighbors in order to be able to compute its next value. Viewing each cell autonomously in this way suggests treating the simulation of a sandpile much like a cellular automata. Every frame, each cell does:

- check if it itself is unstable
- check how many of its neighbors are unstable
- gain a grain for each unstable neighbor and lose 4 grains if it itself was unstable.

Such a view also suggests GPU computation, a technique that has been gaining ground in recent years due to its applicability to highly parallelizable problems. Creating and displaying 3D graphics typically involves a large number of small independent calculations. In particular, computation needs to be done for each pixel on a display (i.e., what color should a pixel be). As such, graphics cards have been developed to handle many small independent calculations very quickly (this can be done by including many small processors on a single card, for example). This ability allows graphics cards to be useful in problems beyond rendering computer graphics. In general, any problem in which many small computations can be performed independently may lend itself to parallelization with GPUs. We have ourselves such a problem in the computation of the stabilizations of sandpiles.

2.1 WebGL sandpiles

The basic principle behind converting the sandpile model to a GPU computation is the translation of sand height into color data in a texture. As images are stored as arrays of color data, we can cast sand heights (and other properties) as color data and instruct the GPU to perform some operations on this data which it can do very quickly when the operations per pixel are independent. This method allows for efficient computation as well as a straightforward way to visualize stabilization.

In the interest of harnessing as much GPU power as possible, we chose to implement the sandpile model using WebGL. WebGL is a derivative of OpenGL—a widely used framework for developing computer graphics—that is designed to render graphics inside a web browser. WebGL makes use of the graphics card of the client (i.e., the computer of the user visiting the website) rather than the server, meaning that as long as web browsers exist supporting WebGL, any computer (and so any existing graphics card) can visit a site using WebGL and run the computations. Improving the speed of a WebGL application is then simply a matter of connecting with a computer containing a more powerful graphics card, as opposed to upgrading the GPU of the server.

The website we created allows the user to simulate the sandpile model using WebGL. For simplicity we focused on simulating the bounded grid graphs discussed above. Various grid sizes can be chosen, and arbitrary amounts of sand can be added to the grid. Configurations can be stabilized and visualized, and the identity can be generated in several ways. The website remains in development and can be found as of this publishing at `http://people.reed.edu/~davidp/web_sandpiles/`. The current source code of the website can be found in the appendix.

We took a "frame-by-frame" approach to stabilization as it is straightforward and leads to interesting visuals. A sandpile configuration is initialized as a texture containing color data for each pixel, representing sand heights, and then is updated and displayed many times per second. In each frame rendered, the GPU applies the rules described above to each cell. This results in animations where all unstable cells in a frame are fired[1].

Useful data besides sand height can also be stored as colors, including whether a cell is a sink, how many times a cell has fired, whether it fired on the previous frame, and so on. This allows for visualization of a variety of aspects of the sandpile model. Of particular interest as we will discuss below is the visualization of the firing vectors

[1]We actually keep two textures, one to represent the next frame to be displayed, and one to represent the current frame. This allows the current configuration to be read and then the new configuration (after applying the firing rule to each cell) to be written to the "next frame" texture. The textures and then swapped and the new "current frame" is displayed.

of stabilizations.

This framework for simulating the sandpile model is flexible and allows for investigation of a number of properties. For example, it is simple to alter the boundary of the grid graph, or to alter the graph by connecting its edges (as on a torus or sphere), or to introduce cells which continually produce new sand ("sources"), or to carry out certain algorithms (such as dropping grains in random locations, as in the experiment mentioned above that reveals the recurrent configurations). While many avenues like these are open for investigation, we chose to focus on the particular problem of quickly generating the identity of a square grid graph.

2.2 Empirical methods

We first implemented generation of the identity by computing the stabilization of kb, where b is the burning configuration, as previously described. Despite the improvements garnered through use of WebGL, we found this method too slow to be practical for larger grids. These experiments however did provide some useful results on how high we should expect k to be given the grid size (Figure 2.1). Fitting a degree 2 polynomial to these data gives us a rough estimate of k for larger grid sizes (Figure 2.2).

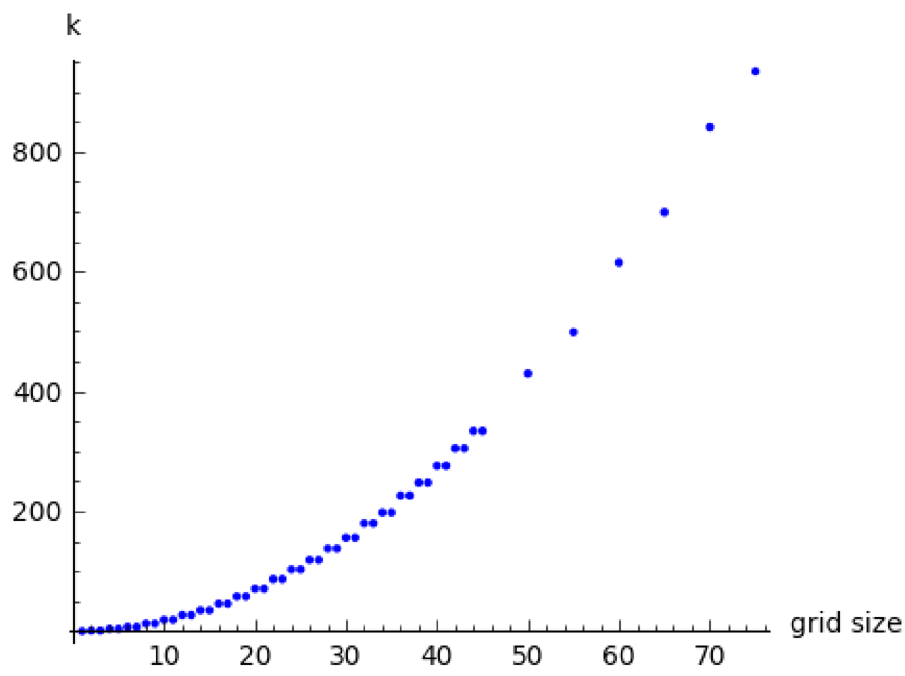

Figure 2.1: Grid size here refers to side length of square grids.

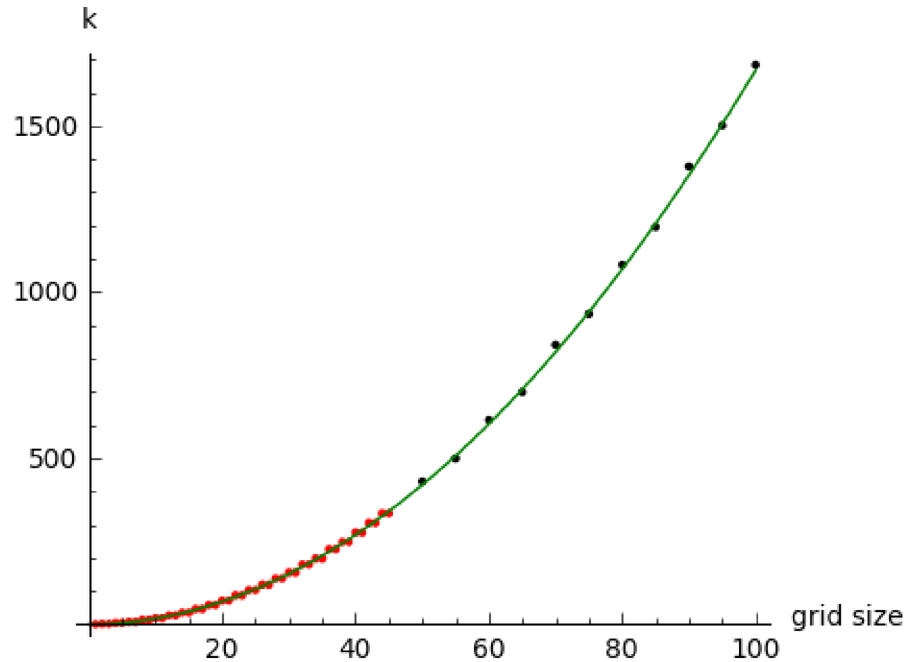

Figure 2.2: The polynomial $ax^2 + bx + c$ was fitted from the red points, and the black points are actual further collected values. The coefficients were a: 0.16574, b: 0.10774, and c: -0.28865.

Estimating this k is useful in two ways. Firstly, stabilizing the configuration kb once is a faster computation in our framework than adding single instances of b, stabilizing, and repeating. Although the same number of total firings occur, the first computation has fewer frames of animation (more cells are fired per frame). Secondly, having an estimate of k gives some idea of how long a computation of the identity may take before attempting it. As Figure 2.3 illustrates, we found it impractical to use this method for grids larger than 500×500.

The basic issue with computing the identity exactly in this way is that, despite whatever improvements in computational speed are made, a large number of calculations still need to be carried out—many frames still need to be stepped through to compute the stabilization. What if we had a way to predict or guess at the identity? Seeing as the identity seems to be scale invariant[2], we have a decent idea of what it "should" look like at different scales (Figure 2.4). However, given the complexity of these images it seems unlikely[3] to be able to predict the patterns for larger grid sizes directly.

Prompted by a suggestion from Wesley Pegden[4], we found an alternative approach through consideration of the previously discussed *firing vectors*.

[2]It is known that the sandpile model exhibits scale invariance in certain circumstances, and a weak limit exists for the identity (Levine, personal communication).

[3]Surely it is not impossible to characterize complex objects like these, but an attempt to do so is beyond our scope.

[4]Personal communication.

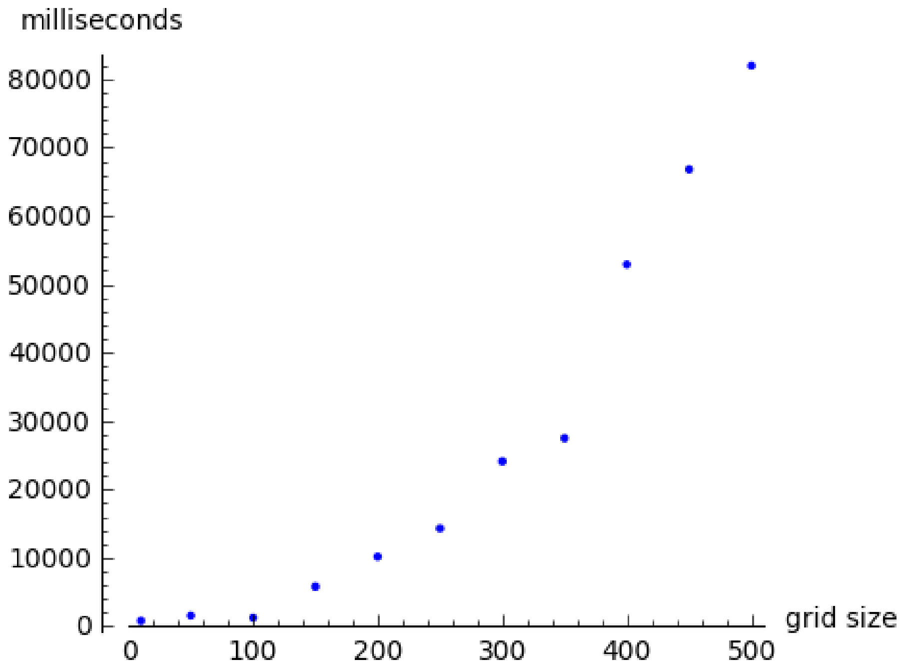

Figure 2.3: Time to compute stab(kb).

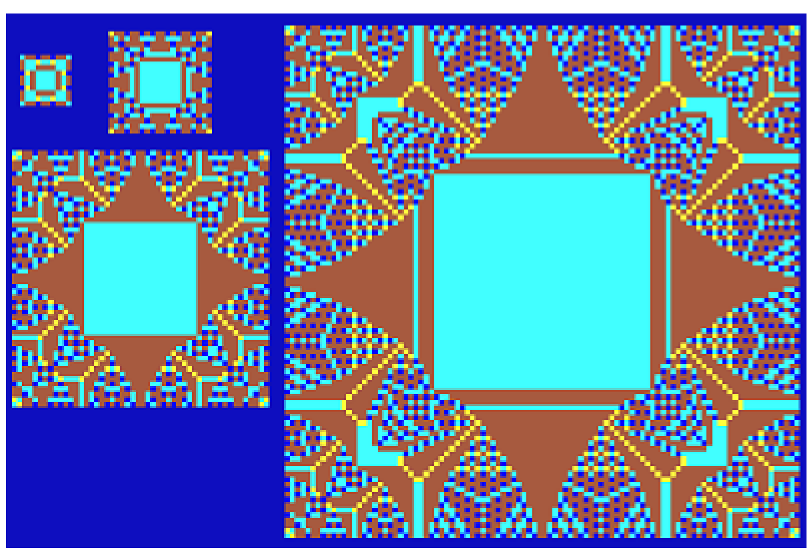

Figure 2.4: The identity on grids of size 10, 20, 50, and 100.

Recall that the identity is equal to $L\sigma_{id}$ for some unique firing vector σ_{id}. We also know that if b is the burning configuration, then $id = \text{stab}(kb) = kb - L\tau$ for some firing script $\tau \geq 0$. Therefore, $\sigma_{id} = k \cdot 1 - \tau$.

Thus to empirically compute σ_{id}, repeatedly fire the sink until the identity is reached and keep track of which cells fired. In doing this for a variety of grid sizes, we noticed that the firing vectors σ_{id} all had very similar shapes (Figure 2.5).

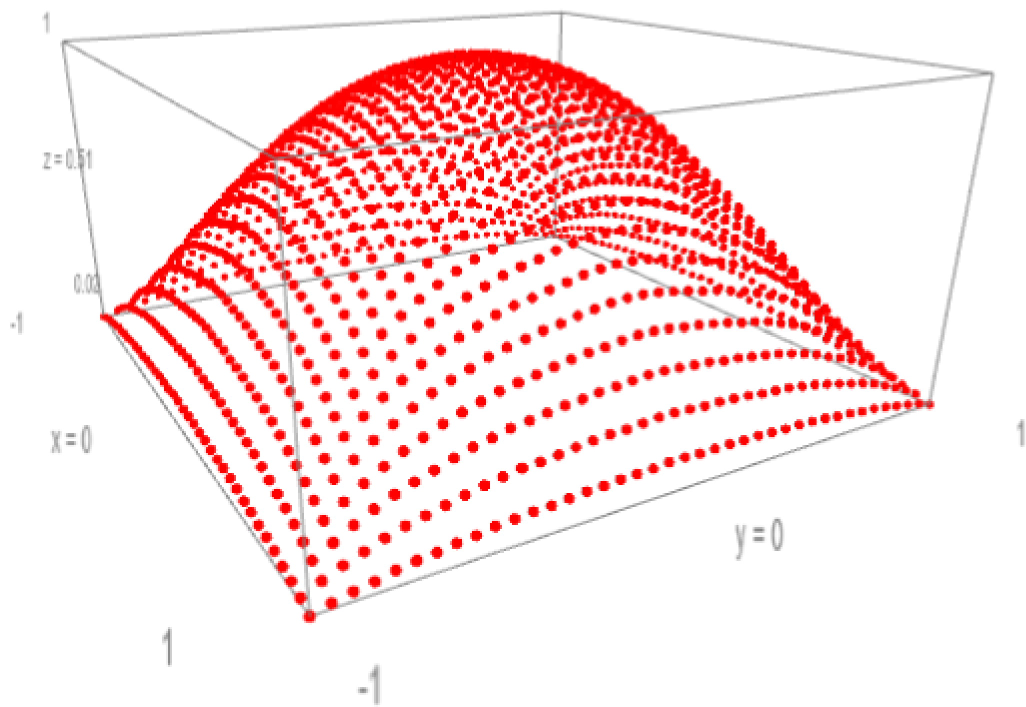

Figure 2.5: The firing vector that gives the identity on a 40×40 grid. This is a plot of the triples (i, j, p) where p is the component of the firing vector with index $(i \cdot 40 + j)$. The (i, j) coordinates have been shifted so that the center is $(0, 0)$ and the values of p have been scaled to lie between 0 and 1.

These surfaces are strikingly simple, especially compared to the complexity of the identity itself! In particular, they exhibit an eight-fold symmetry and resemble a paraboloid or perhaps a multivariate bell curve. We modeled this shape with surfaces exhibiting the same eight-fold symmetry.

In particular, following a suggestion from Ray Mayer, we considered polynomial surfaces of the form $f(x, y) = A + B \cdot (x^2 + y^2) + C \cdot (x^2y^2)$. Even more particularly, we used the following surface, which passes through the points (0, 0, h), (0, 1, s), and (1, 1, c), representing the highest point of the surface, the peak of the side-arcs, and the corners.

$$f(x, y) = h + (s - h) \cdot (x^2 + y^2) + (c + h - 2s) \cdot (x^2y^2)$$

Every firing vector we generated can be characterized by these three points (Table 2.1).

Table 2.1: Empirically determined coefficients

Grid size	h	c	s
2	1	1	1
5	4	2	3
10	19	3	7
15	35	3	10
20	71	4	15
25	103	4	18
30	156	4	23
35	198	4	26
40	276	5	31
45	334	5	34
50	430	5	39
100	1684	6	78
150	3796	6	34
200	6738	7	157
250	10506	7	197
300	15128	8	236
400	26886	8	316
500	41960	9	395
600	60376	9	474
750	94333	9	592
800	107259	9	632
1000	167642	10	790
1200	241378	10	949
1400	328427	10	1107

If such a function accurately describes a firing vector with given h, c, and s, then predicting larger vectors is reduced to predicting these three parameters as a function of the grid size. Testing this requires a suitable notion of "accuracy". As our goal is no more than generating the identity, we chose a certain kind of closeness to the identity as a measure of accuracy of a firing vector. Consider the result of firing a vector generated from the above surface using actual h, c, and s parameters taken from the true firing vector on the 40×40 grid (Figure 2.6).

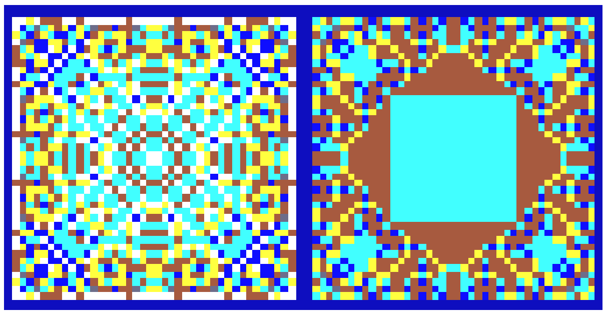

Figure 2.6: The immediate result of firing the vector, followed by its stabilization.

These images are clearly not the identity. However, when we fire the sink, we can see these configurations transition very quickly to the identity:

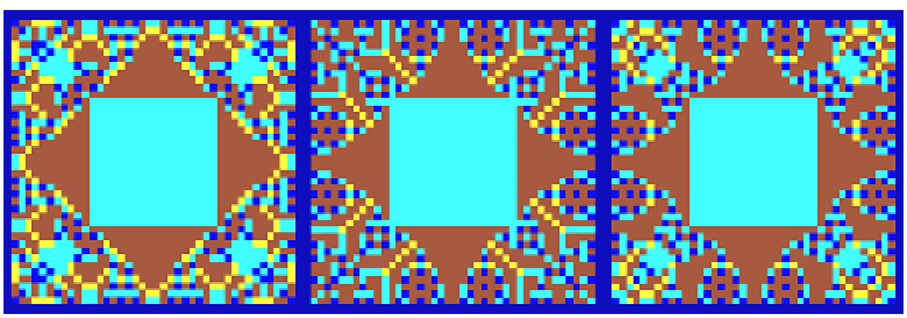

Figure 2.7: Beginning with the configuration from Figure 2.6, fire the sink thrice, then repeat twice (total of 9 sink firings).

Since $L\sigma$ is linearly equivalent to 0, we know that some amount of sink firings bring these estimated identities to the actual identity, and we have noted experimentally that when the estimated firing vector is very close to the true firing vector, this amount will be small (Figure 2.7).

Since the required amount of additional sink firings is easy to determine experimentally, and is useful in that minimizing it minimizes computation, we can use it to measure the fitness of an estimated firing vector. Below is a table showing this value for the surfaces generated from actual h, c, and s values (Table 2.2). We can see that this surface is fairly effective for approximating firing vectors in that it can bring us closer to the identity (i.e. make k smaller). In particular, there is massive improvement from the naive method of firing the sink from the empty configuration without approximating the firing vector.

Table 2.2: k_0 is the number of sink firings needed to reach the identity (from the empty configuration). k_1 is the number of additional sink firings needed after firing the vector estimated using the polynomial surface with coefficients from Table 2.1. k_2 is the number of extra firings needed after firing the least squares fitted surface.

Grid size	k_0	k_1	k_2
2	0	0	0
5	4	0	0
10	19	1	0
15	19	3	1
20	71	3	2
25	103	8	3
30	156	9	3
35	198	15	5
40	276	17	7
45	334	25	11

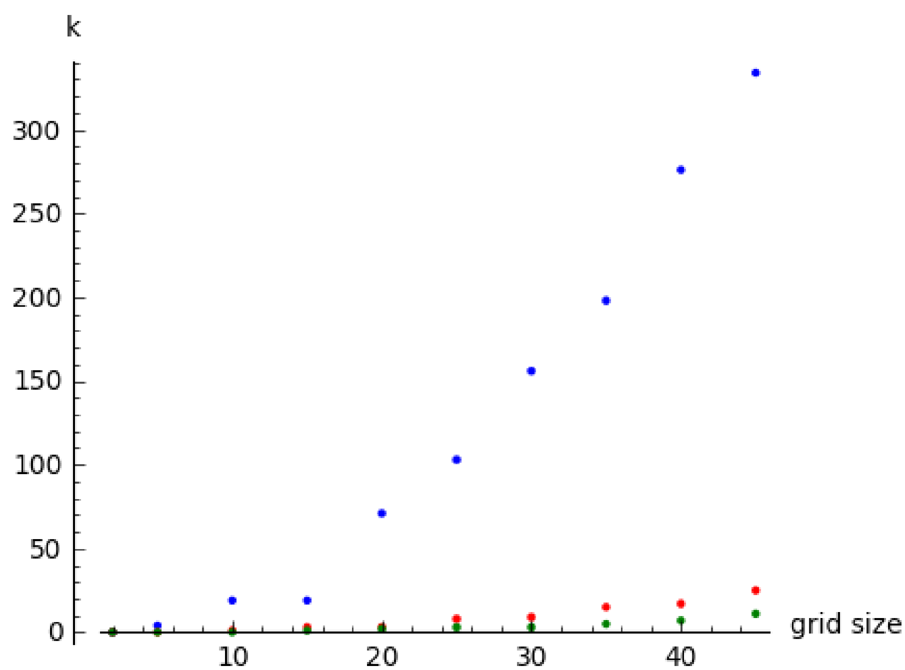

Figure 2.8: Graph of the data from Table 2.2. k_0 is in blue, k_1 is in red, and k_2 is in green.

This surface approximation of the firing vector passes exactly through the h, c, and s points as mentioned. However, it is unclear if that restriction is most useful with respect to this additional sink-firing measure. Consider Figure 2.9. The second surface is the result of fitting the $f(x,y) = h + (s-h)\cdot(x^2+y^2) + (c+h-2s)\cdot(x^2y^2)$ model to the firing vector data directly using a least squares regression. Although this surface does not pass exactly through the h, c, and s points, it more closely approximates the overall shape of the vector. We can use our closeness measure to test which of these two approaches is actually more effective for generating the identity (Table 2.2). Both perform much better than the naive method, and the regression method performs better at least on these particular grid sizes (however the regression method does not at first glance appear "asymptotically" better).

One possibility for exploring the trade-off between the surface passing through particular points and having a better overall fit is to include an additional 'shape' coefficient in the surface function. The following surface passes through the same h, c, and s points when $d = 0$ and features the same eight-fold symmetry:

$$f(x,y) = h + (s-h)\cdot(x^2+y^2) + (c+h-2s-2d)\cdot(x^2y^2) + d\cdot(x^2y^4+x^4y^2)$$

In any case, we would like to predict these coefficients for larger grid sizes. Below are graphs of actual h, c, and s values as a function of grid size (Figures 2.10 – 2.12), along with fitted curves. We can use these predicted coefficients to estimate new firing vectors and then determine their closeness to the identity as above.

Figure 2.13 shows the predicted amount of additional sink firings required after firing the estimated vector obtained from the polynomial surface. We can also take these values into account to further improve our estimate.

In sum, the following improved algorithm computes the identity on an $n \times n$ grid:

- Estimate the coefficients h, c, and s as functions of the grid size using the models shown in Figures 2.10 – 2.12.
- Construct a firing vector σ_{est} by evaluating $f(x,y) = h + (s-h)\cdot(x^2+y^2) + (c+h-2s)\cdot(x^2y^2)$ at integer points with appropriate shifting and scaling[5].
- Fire σ_{est} and stabilize.
- Estimate the number of additional sink firings k_3 using the model shown in Figure 2.13, then fire the sink that many times and stabilize.
- Fire the sink until reaching the identity (a small number of times).

Estimating the firing vector in this way allows us to drastically reduce the number of additional sink-firings needed to reach the identity (compared to beginning with the all 0s configuration).

[5] In particular, we want to create a vector whose $(i\cdot n+j)$th entry contains $p(i,j)$ where $p(i,j) = f(\frac{x-m}{m}, \frac{y-m}{m})$ with $m = \frac{n-1}{2}$ (i.e., we shift and stretch the surface so that $p(m,m) = h$ and $p(0,0) = c$).

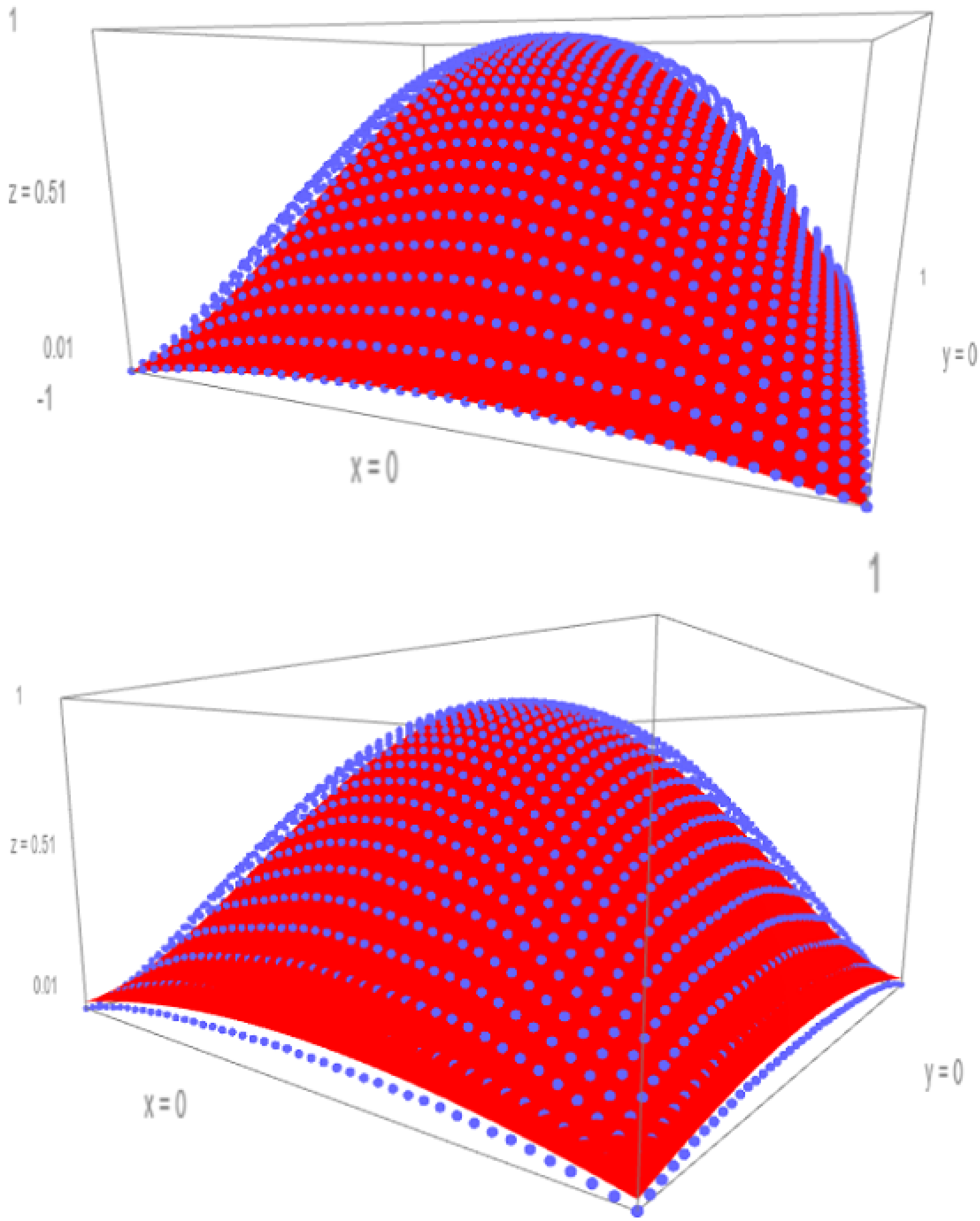

Figure 2.9: The top surface uses exact h, c, and s values collected from σ_{id} for the 45×45 grid. The lower surface was fitted to σ_{id} with least squares. The blue dots are the actual vector σ_{id}.

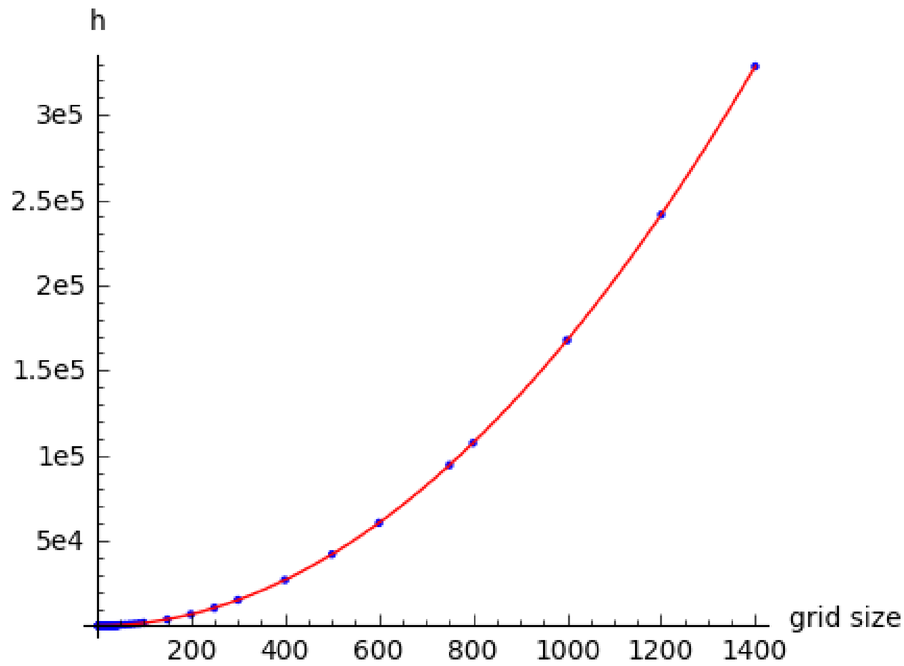

Figure 2.10: h values were modeled as ax^2+bx+c with fitted coefficients $a = 0.16744$, $b = 0.18971$, and $c = -2.7978$.

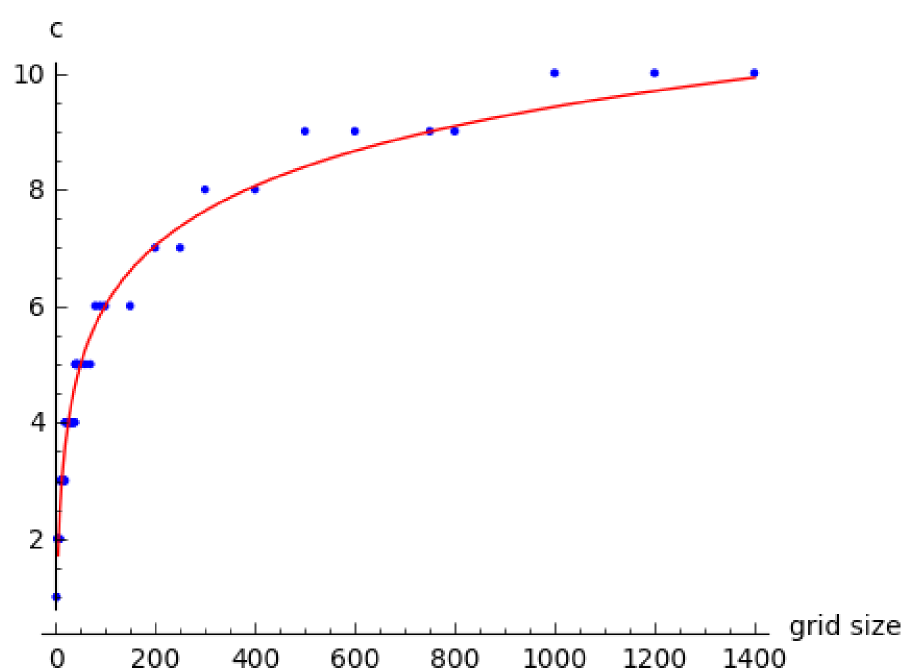

Figure 2.11: c values were modeled as $a+b\cdot log(n)$ with fitted coefficients $a = -0.83617$ and $b = 1.4848$.

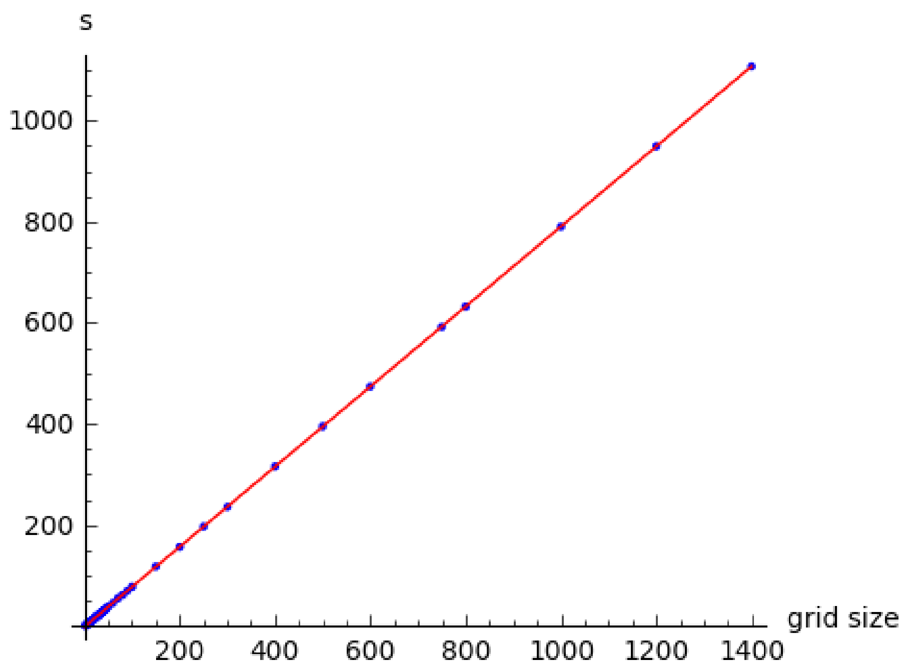

Figure 2.12: s values were modeled as $an + b$ with fitted coefficients $a = 0.79154$ and $b = 0.79154$.

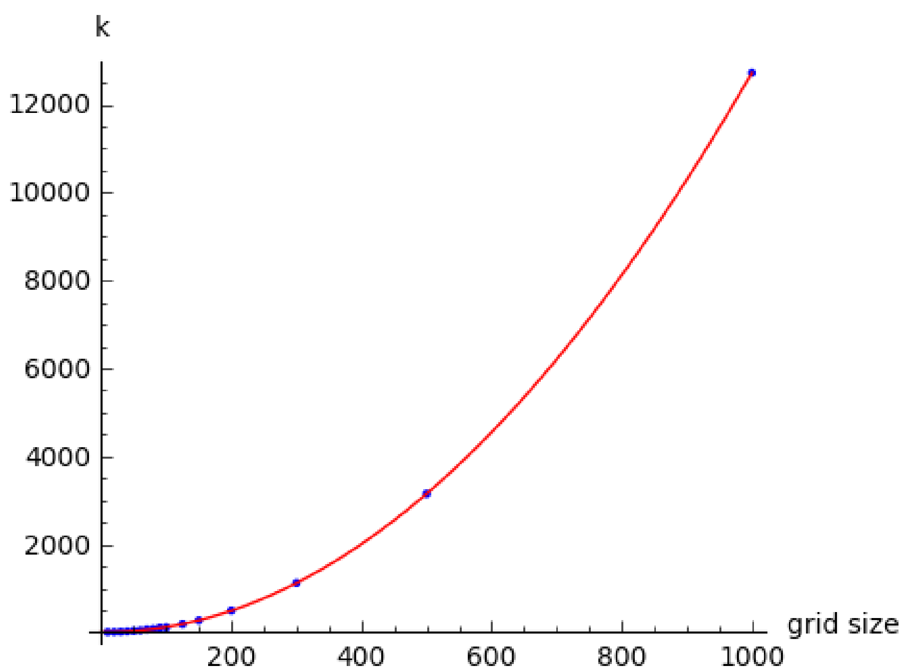

Figure 2.13: k_3 modeled in red as $ax^2 + bx + c$ with fitted coefficients $a = 0.012857$, $b = -0.14120$, and $c = 3.9165$.

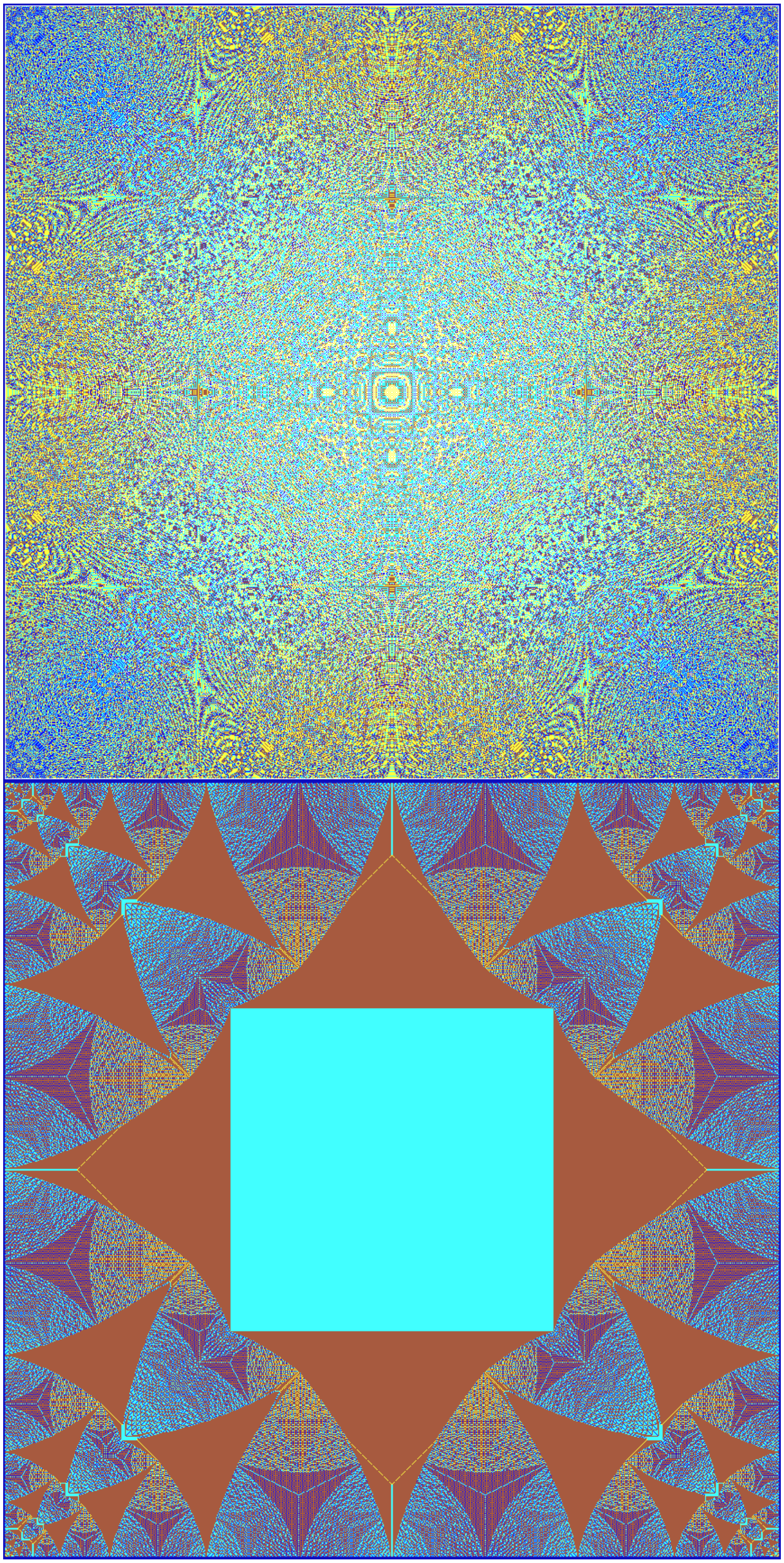

Figure 2.14: The initial firing of an estimated σ_{id} using the polynomial method after estimating h, c, and s, and then its stabilization.

Chapter 3

Results

By estimating coefficients h, c, and s, we generate a firing vector from the surface $f(x, y) = h + (s - h) \cdot (x^2 + y^2) + (c + h - 2s) \cdot (x^2y^2)$, then estimate the required number of additional sink firings, k. Using this method, we were able to achieve extreme closeness to the identity.

Table 3.1: k_0 is the number of sink firings needed to reach the identity (from the empty configuration). The the number of additional sink firings needed after firing the vector estimated using the polynomial surface with predicted coefficients of Figures 2.10 – 2.12 is k_3. The number of further sink firings needed after using the surface method and then predicting and firing k_3 is k_4.

Grid size	k_0	k_3	k_4
10	19	3	0
20	71	5	0
30	156	10	0
40	276	19	0
50	430	30	1
60	615	41	0
70	841	63	6
80	1082	71	0
90	1378	101	6
100	1684	112	0
125	2604	188	1
150	3796	270	0
200	6738	494	4
300	15128	1119	0
500	41960	3146	0
1000	167642	12721	0

There appears to be nearly constant excess required sink firings across grid sizes

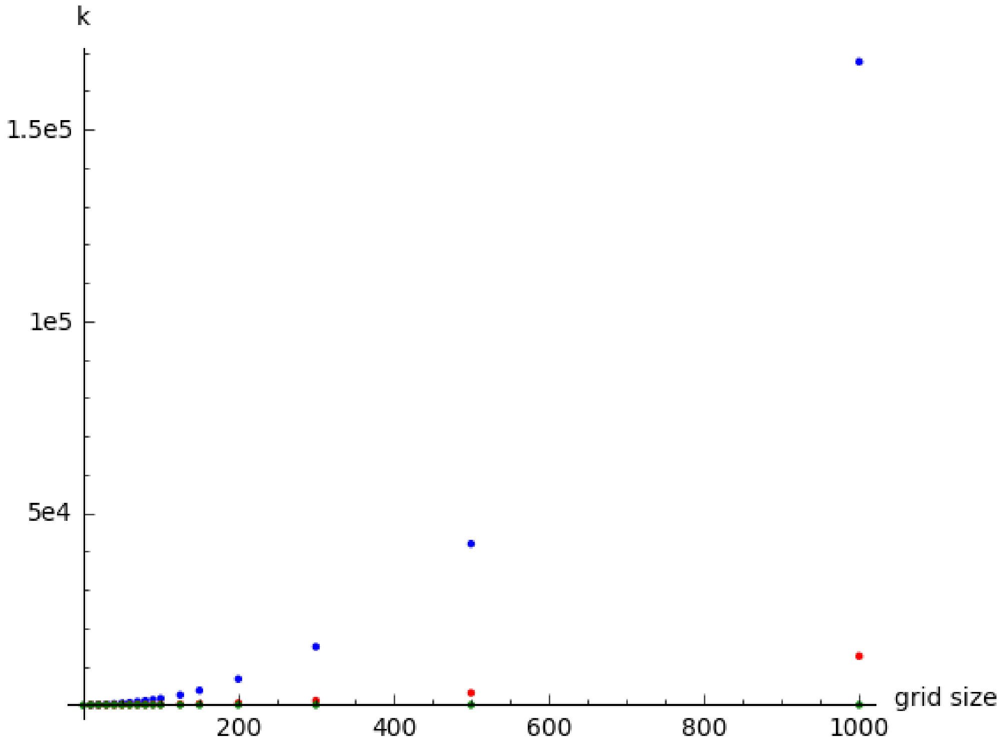

Figure 3.1: Graph of the data from Table 3.1. In blue is k_0, k_3 is in red, and k_4 is in green.

using this method. This is especially nice since we found that one of the most time-intensive operations was repeatedly firing the sink until the identity is reached[1]. If these excess firings k_4 are indeed constant, then in the algorithm we can replace the final "fire the sink until reaching the identity" step with "fire an additional k_4 times". For example, all the k_4s collected above are less than 15, so we can run our algorithm with an extra 15 sink firings. "Overshooting" the identity, while not ideal in terms of optimization, is acceptable, and preferred over the expensive "fire the sink until reaching the identity" step.

While this "closeness to the identity" metric makes sense theoretically, it would be useful to determine if closer estimates indeed translate to faster generation of the identity by computer. We generated the identity for a number of grid sizes using four different methods, and timed their performance.

[1] Because each time, we need to both stabilize, and check if we've reached the identity; a much more expensive operation in total than stabilizing the result of firing the sink k times.

The four methods were:

- Naive method, that is to calculate $\text{stab}((c_{\text{max}} - \text{stab}(2 \cdot c_{\text{max}})) + c_{\text{max}})$.
- $\text{stab}(kb)$, with exact k known from previous data
- $\text{stab}(kb)$, with k estimated from modeling previous data, followed by firing the sink until the identity is reached.
- "Surface" method, that is estimate h, c and s, generate a vector, then estimate further required sink firings (k_3 above) and fire, and lastly fire the sink until the identity is reached (about k_4 more firings).

Note that method 2 is "cheating" in that none of the other methods know k beforehand. So it is not a true method to calculate the identity on any (not previously computed) grid size, but it instead serves as a benchmark for the other methods. If our "surface method" was faster than $\text{stab}(kb)$ even with exact k known, then that would be highly indicative of its usefulness.

Indeed, we see this is the case (Figures 3.2 and 3.3). The "surface" method performs better than any other at every tested grid size. Moreover, the runtime for both the naive and burning configuration methods appears to be growing very quickly, while the surface method has a much gentler slope. We also noted during the performance of these tests that when attempting even higher grid sizes with the surface method, memory became an issue before runtime did. That is, the limiting factor became the space to store the grid, rather than the time to execute computations on the grid. This is in contrast to, for example, the naive method, which quickly becomes temporally infeasible above grids of around size 1000 in addition to the memory issues.

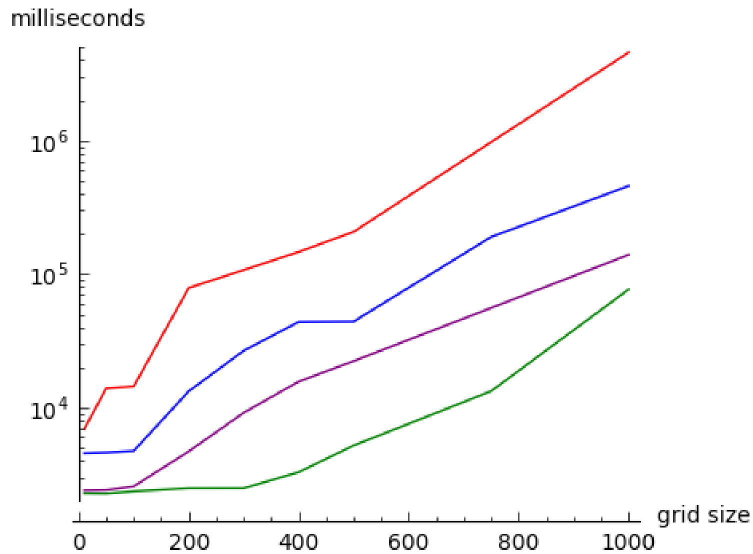

Figure 3.2: Runtime of the naive method (blue), the exact k method (purple), the estimate k method (red) and the surface method (green). The milliseconds axis is plotted on a log scale. These tests were performed using a NVIDIA GeForce GTX 950 GPU (2 GB memory, 768 cores).

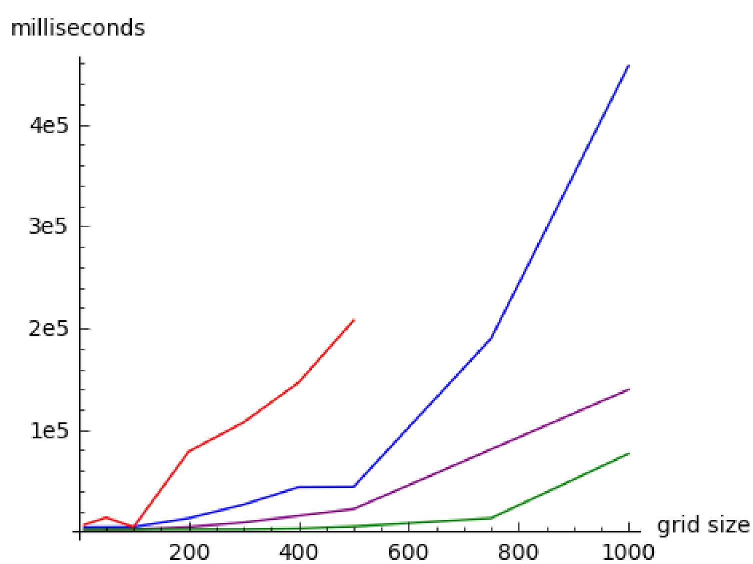

Figure 3.3: Runtime of the naive method (blue), the exact k method (purple), the estimate k method (red) and the surface method (green). The extremely large value (4,580,229) for the "estimate k" method at grid size 1000 is omitted for scale.

Conclusion

In this project we focused on developing faster methods of computing large sandpiles. We used GPU computing as a new framework for performing the computations in the first place, as well as developed methods of quickly computing the identity element on grid graphs.

Overall, we found the methods of computing $\text{stab}((c_{\max} - \text{stab}(2 \cdot c_{\max})) + c_{\max})$ and of computing $\text{stab}(kb)$ for large k to be inadequate for grid graphs larger than around 500×500. In addition, we found estimating the firing vector σ_{id} (such that $L\sigma_{id} = id$) to be a fruitful approach, with drastic improvements in both runtime and distance to the identity.

This general approach could be altered and possibly improved by using different particular approximations of σ_{id}. We chose to use a polynomial surface with eight-fold symmetry which passes through a particular set of points, but a better approximation likely exists, involving perhaps more parameters or a different type of surface. Other routes to the identity are possible as well. For example, given that $L\sigma_{id} = id$ for some firing vector σ_{id}, one could determine σ_{id} by computing $L^{-1}id$, which may be easier than finding or estimating σ_{id} directly. Another option would be to attempt to predict τ where $\text{stab}(kb) = (kb) - L\tau$, which again may turn out to be easier than predicting σ_{id}.

The framework and methods developed in this project can be easily adapted to a number of future interesting problems. In particular, it would be interesting to investigate the behavior of the sandpile model on non-square grids (we previously noted that the identity even on non-square grid exhibits some of the familiar fractal features), or the effects of the addition of different kinds of cells (one could introduce "source" cells which constantly produce sand, for example), or the effects of connecting certain non-adjacent cells (i.e., changing the graph. We can run the simulation on a torus, for example.).

It would also be useful to further develop the graphical representation of the sandpiles. WebGL provides tools to create general computer graphics (in particular, 3D graphics), and so the sandpiles could be visualized in 3D, or run on polyhedra, etc. Since any graph can be embedded in $\mathbb{R}^3$, one interesting possibility is to display any given graph in 3D space, and run the sandpile simulation with nodes colored by sand heights. However, any such generalization of the GPU computation method to more general (non-grid) graphs would require major restructuring of the application.

The study of the dynamics of sandpiles is another area in which our application may be useful. While most of our focus has been on manipulating and computing particular stable configurations, our application naturally allows us to display animations of any number of operations, such as stabilization. It is difficult not to imagine waves or avalanches when viewing these animations, and we feel the playful nature of the application (being able to click around and draw, adding sand anywhere) is especially conducive to exploration of sandpile dynamics. This in part motivated our choice to develop an online application, so that many may view it and explore sandpiles for themselves.

As mentioned, the WebGL application remains in development, but we have included full code of the current iteration in the appendix. Our aim going forward is to further improve the methods developed here and to explore new possibilities afforded by the power of GPU computing. We also hope to continue creating these intricate sandpiles and in so doing perhaps assist in illuminating their structure and behavior.

Appendix: Code

The code of the sandpile simulation website is divided into three main pieces: the HTML for the webpage itself, the Javascript code that is run by the HTML, and the shader code written in GLSL which is run by Javascript in order to carry out WebGL instructions.

The first files are *sand.frag*, *draw.frag*, *copy.frag*, and *quad.vert*. *sand.frag* gives the core automata firing rules, and is run on the back texture once per frame, advancing the simulation. The color values in the cells of the back texture are only data. *draw.frag* reads the back texture and displays actual colors on the front texture to the viewer, and allows for customization of the display. Included in *draw.frag* are a variety of options for color schemes, one of which (named "Wesley" in honor of Wesley Pegden who we first saw use these colors) is used in the images provided throughout this thesis.

copy.frag has minor use, allowing one texture to be copied to another.

quad.vert is a vertex shader establishing the geometry to which the fragment shaders are applied. In our case the geometry is just a flat plane, but it can be transformed if we wish with projection matrices. We do not make much use of this in the project, so it is an area of possible exploration.

```
// sand.frag
#ifdef GL_ES
precision highp float;
#endif

uniform sampler2D state;

uniform vec2 scale;
uniform vec2 res;

int max = 1048576 - 1;
vec2 center = vec2(.5, .5);

// data is stored in RBGA float channels
// r : sand height
// g : cell type, 0 = node, 1 = sink, 2 = source, 3 = wall
// b : two bits for "fired last round?" and "negative or positive sand?"
// a : total firings at this cell so far (since last reset)

// below are just some helper functions

// decode and encode color data and sand heights
ivec4 decode (vec4 data){
        return ivec4(floor( .5 + float(max) * data.r), floor(.5 + float(max) * data.g), floor(.5 +
            float(max) * data.b), floor(.5 + float(max) * data.a));
}

```

```
vec4 encode (ivec4 data){
        return vec4(float(data.r)/float(max), float(data.g)/float(max), float(data.b)/float(max),
             float(data.a)/float(max));
}

ivec4 get(int x, int y){ //lookup at current spot with some pixel offset
        return decode(texture2D(state, (gl_FragCoord.xy + vec2(x, y)) / scale));
}

int tens(int n){
        return int(floor(float(n)/float(10)));
}

int ones(int n){
        return n - 10*tens(n);
}

// main is executed for each pixel in the state texture once per frame (once per call of sand.step()
     in the javascript).

void main() {
        vec2 position = gl_FragCoord.xy;
        float x = position.x;
        float y = position.y;

        int N, E, W, S, C, F;
        int deg = 4; //this is just for walls, I subtract from this when adjacent to a wall
        ivec4 cell = get(0,0);
        ivec4 cellN = get(0,1);
        ivec4 cellE = get(1,0);
        ivec4 cellW = get(-1,0);
        ivec4 cellS = get(0,-1);
        vec4 result;

        if (cell.g == 0){
                result = encode(ivec4(0,0,0,0));
        } else if (cell.g == 3){
                result = encode(ivec4(0,3,0,0));
        } else {
                // determine outdegree (I'm treating walls as the edge to that node being deleted)
                if (cellN.g == 3){deg--;}
                if (cellE.g == 3){deg--;}
                if (cellS.g == 3){deg--;}
                if (cellW.g == 3){deg--;}

                // checking if a neighbor fired last round (or if a neighbor is a source), in which
                     case we get one

                if (tens(cellN.b) == 1 || cellN.g == 2){N = 1;} else {N = 0;}
                if (tens(cellE.b) == 1 || cellE.g == 2){E = 1;} else {E = 0;}
                if (tens(cellS.b) == 1 || cellS.g == 2){S = 1;} else {S = 0;}
                if (tens(cellW.b) == 1 || cellW.g == 2){W = 1;} else {W = 0;}

                // these two parts below are the core of the cellular automata loop described in the
                     computation section of the paper

                // if I will fire
                if (cell.r >= deg) {C = -deg; F = 1;} else {C = 0; F = 0;}

                // how much sand I get from neighbors
                if (ones(cell.b) == 1){
                        if (N + E + S + W + C - cell.r >= 0){
                                cell.r = (N + E + S + W + C) - cell.r;
                                cell.b = tens(cell.b);
                        } else {
                                cell.r = -1*(N + E + S + W + C - cell.r);
                                cell.b = tens(cell.b) + 1;
                        }
```

```
			} else {
				cell.r = (N + E + S + W + C) + cell.r;
			}

			cell.a += F;                              // total firings
			cell.b = ones(cell.b) + 10*F; // fired this time?

			result = encode(cell);
		}


	gl_FragColor = result;
}
```

```
// draw.frag
#ifdef GL_ES
precision highp float;
#endif

uniform vec2 scale;
uniform vec2 shift;
uniform sampler2D state;
uniform float color;

int max = 1048576 - 1;

int color_choice = int(color);

ivec4 decode (vec4 data){
	return ivec4(floor(.5 + float(max) * data.r), floor(.5 + float(max) * data.g), floor(.5 + float(max) * data.b), floor(.5 + float(max) * data.a));
}

vec4 encode (ivec4 data){
	return vec4(float(data.r)/float(255), float(data.g)/float(255), float(data.b)/float(255), float(data.a)/float(255));
}

ivec4 get(int x, int y){ //lookup at current spot with some pixel offset
	return decode(texture2D(state, (gl_FragCoord.xy + vec2(x, y) + shift) / scale ));
}

int hundreds(int n, int base){
	return int(floor(float(n)/float(base*base)));
}

int tens(int n, int base){
	return int(floor(float(n)/float(base)));
}

int ones(int n, int base){
	return n - 10*tens(n, base);
}

vec4 color_select(ivec4 cell, int select, int sinks, int sources){
	ivec4 result;

	if (select == 0){
		int size = int(abs(float(cell.r)));

		//wesley colors

		if (size == 0){
			result = ivec4(0,0,255,0);   //dark blue
		} else if (size == 1){
			result = ivec4(255,255,0,0); //yellow
```

```
        } else if (size == 2){
            result = ivec4(51,255,255,0); //light blue
        } else if (size == 3){
            result = ivec4(153,76,0,0);  //brown
        } else if (size >= 4){
            result = ivec4(255,255,255,0); //white
        }

        if (cell.r < 0) {
            result = ivec4(100) - result;
        }

    } else if (select == 1){
        int size = int(abs(float(cell.r)));

        //this scheme for the numberphile video

        if (size == 0){
            result = ivec4(10,10,100,0); //black
        } else if (size == 1){
            result = ivec4(255,255,0,0); //yellow
        } else if (size == 2){
            result = ivec4(0,0,255,0);   // blue
        } else if (size == 3){
            result = ivec4(255,0,0,0);   //red
        } else if (size >= 4){
            result = ivec4(255,255,255,0); //white
        }

        result = ivec4(result.r, result.g, result.b, 0);

        if (cell.r < 0) {
            result = ivec4(255) - result;
        }

    } else if (select == 2){

        // shows if something fired last time

        if (cell.b == 0){
            result = ivec4(50,50,50,0);
        } else {
            result = ivec4(255,255,255,0);
        }


    } else if (select == 3){

        //this scheme shows unstable vertices

        if (cell.r == 4) {
            result = ivec4(255,255,255,0);
        } else {
            result = ivec4(50,50,50,0);
        }

    } else if (select == 4){

        //shows how many times a cell has fired (256^3 colors)
        int size = int(abs(float(cell.a)));
        int base = 10; //must be 0 < base < 256

        result = ivec4(ones(size, base)*(300/base), tens(size, base)*(255/base),
            hundreds(size, base)*(255/base), 0);

        if (cell.a < 0) {
            result = ivec4(255) - result;
        }
```

```
        } else if (select == 5){
                //multiplicative gradient (256*3 colors)
                int size = int(abs(float(cell.r)));
                int base = 10; //must be 0 < base < 256

                if (size < base * 1) {
                        result = ivec4(0, 0, size*(255/base), 0);
                } else if (size < base * 2) {
                        result = ivec4(0, (size - base)*(128/base), 255, 0);
                } else {
                        result = ivec4((size - base - base) *(64/base), 255, 255, 0);
                }

                if (cell.r < 0) {
                        result = ivec4(255) - result;
                }

        } else if (select == 6){
                int size = int(abs(float(cell.r)));
                //exponential gradient (256^3 colors)

                int base = 10; //must be 0 < base < 256

                result = ivec4(ones(size, base)*(255/base), tens(size, base)*(255/base),
                     hundreds(size, base)*(255/base), 0);

                if (cell.r < 0) {
                        result = ivec4(255) - result;
                }
        }


        if (cell.g == 0){
                result = ivec4(0,0,128,0);
        } else if (cell.g == 2){
                result = ivec4(0,255,0,0);
        } else if (cell.g == 3){
                result = ivec4(255,0,0,0);
        }

        //can add as many color schemes as you'd like
        return encode(result);
}

void main() {
        gl_FragColor = color_select(get(0,0), color_choice, 0, 0);
}
```

```
// copy.frag
#ifdef GL_ES
precision mediump float;
#endif

uniform sampler2D state;
uniform vec2 scale;

void main() {
        gl_FragColor = texture2D(state, gl_FragCoord.xy / scale);
}
```

```
// quad.vert
#ifdef GL_ES
precision highp float;
#endif
```

```

attribute vec2 quad;

uniform vec3 matrix1;
uniform vec3 matrix2;
uniform vec3 matrix3;

void main() {
	mat3 matrix = mat3(matrix1, matrix2, matrix3);
	gl_Position = vec4((matrix*vec3(quad, 1)).xy, 0, 1.0);
}
```

Next we have the HTML for the webpage. This file simply provides the canvas which we will draw to with Javascript and WebGL. The chosen width and height are the "actual" width and height of the canvas, putting a bound on how large of a sandpile can be run. The canvas as displayed to the client will fill the screen, or can otherwise have a custom apparent resolution.

The included Igloo script is a wrapper for some of the WebGL commands used in the *sand.js* file. It was created by Christopher Wellons, whose Game of Life implementation using WebGL was an invaluable source of guidance and inspiration during this project. His live implementation can be found at `http://nullprogram.com/webgl-game-of-life/` with the source at `https://github.com/skeeto/webgl-game-of-life/`.

```
// index.html
<!DOCTYPE html>
<html>
	<head>
		<title>WebGL Sandpile</title>
		<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
		<link rel="stylesheet" href="gol.css"/>
		<script src="lib/igloo-0.0.3.js"></script>
		<script src="lib/jquery-2.1.1.min.js"></script>
		<script src="js/sand.js"></script>
	</head>
	<body>
		<canvas id="sand" width="2100" height="2100"></canvas>
	</body>
</html>
```

Lastly, we have the longest file, *sand.js*, which does most of the work of running the website. Many functions are included which allow for a number of different user interactions with the sandpile, not all of which are currently used in the live website. The most important pieces are the *step* and *draw* functions, which call on the various **.frag* files to carry out the simulation of the sandpile. These functions alternate on a timer, displaying the animation to the canvas.

```
// sand.js
const max = 1048576 - 1;

// this function is run at the bottom to initialize the sandpile simulation
function SAND(canvas, scale) {
	// initialize webgl and some variables
	var gl = this.gl = canvas.getContext('webgl', {preserveDrawingBuffer: true});
	if (gl == null) {
		alert('Could not initialize WebGL!');
		throw new Error('No WebGL');
	}
```

```
        gl.getExtension('OES_texture_float');

        scale = this.scale = 2;
        this.w = canvas.width;
        this.h = canvas.height;
        this.viewsize = vec2(this.w, this.h);
        this.viewx = 0;
        this.viewy = 0;
        this.dx = 100;
        this.dz = 300;
        this.statesize = vec2(this.w / scale, this.h / scale);
        this.timer = null;
        this.lasttick = SAND.now();
        this.fps = 0;

        this.d = 200.0;
        this.m = this.d;
        this.n = this.d;
        this.res = vec2(this.m, this.n);

        this.shift = vec2(-600,50);

        this.saves = [];
        this.save_id = 0;
        this.user_saves = 0;

        this.firing_vectors = [];
        this.firing_vector_id = 0;

        this.shape_choice = 1; //default to square

        this.identity = null;

        this.brush_height = 0;
        this.brush_type = 0;

        this.speed = 1;
        this.frames = 1;
        this.color = 0.0;

        gl.disable(gl.DEPTH_TEST);

        this.programs = {
                copy: new Igloo.Program(gl, 'glsl/quad.vert', 'glsl/copy.frag'),
                sand: new Igloo.Program(gl, 'glsl/quad.vert', 'glsl/sand.frag'),
                draw: new Igloo.Program(gl, 'glsl/quad.vert', 'glsl/draw.frag')
        };

        this.buffers = {
                quad: new Igloo.Buffer(gl, new Float32Array([
                        -1, -1, 1, -1, -1, 1, 1, 1
                ]))
        };

        this.textures = {
                front: this.texture(),
                back: this.texture()
        };

        this.framebuffers = {
                step: gl.createFramebuffer()
        };

        // selects initial shape (square in this case) and palces initial sand (none in this case)

        this.set_surface(this.shape_choice);
        this.set(this.fullstate(0));

```

```
	// all these below create the interface buttons and forms

	var toolbar = document.createElement( 'div' );
	toolbar.style.position = 'absolute';
	toolbar.style.top = '25px';
	toolbar.style.left = '25px';
	document.body.appendChild( toolbar );

	var rightside = document.createElement( 'div' );
	rightside.style.cssFloat = 'left';
	toolbar.appendChild( rightside );

	add_form(toolbar, "inspect_val", "1", 'Inspect', f = function() {
		sand.brush_type = 6;
	});

	add_form(toolbar, "full_field", "4", 'Set each cell to n', f = function() {
		sand.set(sand.fullstate($("#full_field").val()));
	});

	add_form(toolbar, "arithmetic_field", "4", 'Add n to each cell', f = function() {
		sand.plus($("#arithmetic_field").val());
		sand.draw();
	});

	var save_div = document.createElement( 'div' );
	save_div.setAttribute('id', 'saves');

	var adds_div = document.createElement( 'div' );
	adds_div.setAttribute('id', 'adds');

	add_form(toolbar, "fire_sink_field", "1", 'Fire sink k times', f = function() {
		sand.fire_sink($("#fire_sink_field").val());
		sand.canvas.focus();
	});

	add_form(toolbar, "height_field", "1", 'Set clicked cells to n', f = function() {
		sand.brush_height = ($("#height_field").val());
		sand.brush_type = 4;
	});

	br(toolbar);
	add_form(toolbar, "save_field", "my sandpile", 'Save state', f = function() {
		sand.save();
		sand.user_saves += 1;

		var newButton = document.createElement("input");
		newButton.type = "button";
		newButton.id = sand.save_id - 1;
		newButton.value = "load " + ($("#save_field").val());
		newButton.onclick = function(){
			sand.load(newButton.id);
		};
		document.getElementById("saves").appendChild(newButton);

		var newButtonAdd = document.createElement("input");
		newButtonAdd.type = "button";
		newButtonAdd.id = sand.save_id - 1;
		newButtonAdd.value = "add " + ($("#save_field").val());
		newButtonAdd.onclick = function(){
			sand.set(sand.add(sand.saves[newButtonAdd.id], sand.get()));
		};
		document.getElementById("adds").appendChild(newButtonAdd);
	});

	toolbar.appendChild(save_div);
	toolbar.appendChild(adds_div);
```

```
        var firing_vectors_div = document.createElement( 'div' );
        firing_vectors_div.setAttribute('id', 'firing_vectors');
        add_form(toolbar, "save_firing_vector_field", "my vector", 'Save firing vector', f =
            function() {
                sand.save_firing_vector();
                var newButton = document.createElement("input");
                newButton.type = "button";
                newButton.id = sand.firing_vector_id - 1;
                newButton.value = "fire " + ($("#save_firing_vector_field").val());
                newButton.onclick = function(){
                        sand.fire_vector(sand.firing_vectors[newButton.id]);
                };
                document.getElementById("firing_vectors").appendChild(newButton);
        });
        toolbar.appendChild(firing_vectors_div);

        add_form(toolbar, "name_field", "my sandpile", 'Download state', f = function() {
                var state = sand.get();
                download("data:text/csv;charset=utf-8," + state, $( "#name_field").val() + ".txt");
        });

        add_form(toolbar, "speed_field", "1", 'Frames per millisecond', f = function() {
                sand.set_speed($( "#speed_field" ).val(), $( "#delay_field" ).val());
                sand.draw()
        });

        add_form(toolbar, "delay_field", "1", 'Milliseconds per frame', f = function() {
                sand.set_speed($( "#speed_field" ).val(), $( "#delay_field" ).val());
                sand.draw()
        });

        add_form(toolbar, "run_field", "100", 'Run for n steps', f = function() {
                sand.run($( "#run_field" ).val());
                sand.draw()
        });

        add_button(rightside, 'Time burning config method', f = function() {
                sand.time_burning_config_method();
        });

        //brush tools
        add_button(rightside, 'Add single grains', f = function() {
                sand.brush_type = 0;
        });

        add_button(rightside, 'Add sinks', f = function() {
                sand.brush_type = 1;
        });

        add_button(rightside, 'Add sources', f = function() {
                sand.brush_type = 2;
        });

        add_button(rightside, 'Add walls', f = function() {
                sand.brush_type = 3;
        });

        add_button(rightside, 'Fire', f = function() {
                sand.brush_type = 5;
        });

        add_button(rightside, 'Random Stable Configuration', f = function() {
                sand.setRandom();
                sand.draw();
        });

        add_form(toolbar, "size_field", this.d, 'Choose grid size', f = function() {
                var n = ($("#size_field").val());
```

```
		if (n < sand.w/sand.scale){
			sand.m = n;
			sand.n = n;
			sand.res.x = n;
			sand.res.y = n;
			sand.reset();
			sand.set_surface(1);
		} else {
			alert("Please choose a smaller grid. Max is " + (sand.w/sand.scale - 1) + ".");
		}
	});

	add_form(toolbar, "state_val", "", 'Get state', f = function() {
		$("#state_val").val(sand.get());
	});

	add_form(toolbar, "firings_val", "", 'Get total firings', f = function() {
		var gl = sand.gl;
		var state = sand.get();
		var n = 0;

		for (var i = 0; i < state.length; i += 4){
			n += state[i + 3];
			//alert(n)
		}

		//alert(n);
		$("#firings_val").val(n);
	});

	add_form(toolbar, "vector_val", "", 'Get firing vector', f = function() {
		var vec = sand.get_firing_vector(sand.get());
		$("#vector_val").val(vec);
		copyToClipboard(vec);
	});

	add_button(rightside, 'get h, c, s', f = function() {
		var vec = sand.get_firing_vector(sand.get());
		alert([vec[(sand.m/2)*(sand.m) + (sand.m/2)], vec[0], vec[sand.m/2]]);
	});

	br(rightside);
	add_button(rightside, 'Calculate Identity', f = function() {
		sand.set_identity();
	});

	add_button(rightside, 'Approximate k', f = function() {
		$("#fire_sink_field").val(sand.approx_k());
	});

	add_button(rightside, 'Approximate Identity', f = function() {
		var n = sand.n;
		var m = sand.m;
		if (n == m){
			//alert('This may take a while');
			sand.reset();
			v = sand.approx_identity_4(n);
			sand.fire_vector(v);
			$("#vector_val").val(v);
		} else {
			alert("This function not yet implemented for nonsquare grids")
		}
	});

	add_button(rightside, 'Fire sink until identity', f = function() {
		alert(sand.fire_sink_until_id());
	});
```

```
        add_button(rightside, 'Approximate Identity Algorithm', f = function() {
                var n = sand.n;
                var m = sand.m;
                if (n == m){
                        //alert('This may take a while');
                        //alert(m)
                        sand.reset();
                        var t0 = performance.now();
                        sand.approx_identity_alg(n);
                        var t1 = performance.now();
                        alert("Calculation took " + (t1 - t0) + " milliseconds.")
                } else {
                        alert("This function not yet implemented for nonsquare grids")
                }
        });

        add_form(toolbar, "d_field", "0", 'Approx identity with certain d', f = function() {
                var n = sand.n;
                sand.reset();
                var t0 = performance.now();
                sand.approx_identity_alg(n, $("#d_field").val());
                var t1 = performance.now();
                alert("Calculation took " + (t1 - t0) + " milliseconds.")
        });

        br(rightside);

        add_button(rightside, 'Stabilize', f = function() {
                sand.stabilize();
        });

        add_button(rightside, 'Dualize', f = function() {
                sand.dualize();
        });
        add_button(rightside, 'Reset', f = function() {
                sand.reset();
        });
        add_button(rightside, 'Clear firing vector', f = function() {
                sand.clear_firing_history();
                sand.draw();
        });
        br(rightside);
        add_button(rightside, 'Add a random grain', f = function() {
                sand.set(sand.add_random(sand.get()));
                sand.draw();
        });

        add_button(rightside, 'Calculate recurrent inverse of current state', f = function() {
                sand.rec_inverse();
                sand.draw();
        });
        add_form(toolbar, "fire_field", "my vector", 'Fire a vector', f = function() {
                sand.fire_vector($("#fire_field").val().split(",").map(Number));
        });

        add_form(toolbar, "paste_field", "my state", 'Load a state', f = function() {
                sand.set($( "#paste_field" ).val().split(",").map(Number));
                sand.draw()
        });

        var colors = [['Wesley', 0],['Luis', 1],['Which just fired', 2],['Unstable cells',
            3],['Firing vector', 4],['256*3 colors', 5],['256^3 colors', 6]];
        add_select(toolbar, colors, f = function(e) {
                sand.color = e.target.value;
        });
}
```

```
// helper functions in creating the interface

function br(parent){
        var blank = document.createElement("br");
        parent.appendChild(blank);
}

function add_select(parent, options, selectfunc){
        var select = document.createElement( ’select’ );
        for (var i = 0; i < options.length; i++) {
                var option = document.createElement(’option’);
                option.textContent = options[i][0];
                option.value = options[i][1];
                select.appendChild(option) ;
        }

        select.addEventListener( ’change’, function (event) {
                selectfunc(event);
                f.blur();
        });

        parent.appendChild(select);
}

function add_button(parent, buttontext, buttonfunc){
        var f = document.createElement(’button’);
        f.textContent = buttontext;
        f.addEventListener(’click’, function(event){
                event.preventDefault();
                buttonfunc();
                f.blur();
        });
        parent.appendChild( f );
}

function add_form(parent, fieldname, fieldval, buttontext, buttonfunc){
        var f = document.createElement(’form’);

        var i = document.createElement("input");
        i.setAttribute(’type’,"text");
        i.setAttribute(’id’,fieldname);
        i.setAttribute(’value’,fieldval);

        var s = document.createElement(’button’);
        s.setAttribute(’type’,"submit");
        s.textContent = buttontext;

        f.addEventListener(’submit’, function(event){
                event.preventDefault();
                buttonfunc(fieldname);
                i.blur();
        });

        f.appendChild( i );
        f.appendChild( s );
        parent.appendChild( f );
}

// allows resizing the browser window

function resize(canvas) {
  var displayWidth = canvas.clientWidth;
  var displayHeight = canvas.clientHeight;

  if (canvas.width != displayWidth || canvas.height != displayHeight) {
    canvas.width = displayWidth;
    canvas.height = displayHeight;
  }
```

```
}

SAND.now = function() {
  return Math.floor(Date.now() / 1000);
};

// swap, step, and draw are the core of all this

SAND.prototype.swap = function() {
        var tmp = this.textures.front;
        this.textures.front = this.textures.back;
        this.textures.back = tmp;
        return this;
};

SAND.prototype.step = function() {
        if (SAND.now() != this.lasttick) {
                $('.fps').text(this.fps + ' FPS');
                this.lasttick = SAND.now();
                this.fps = 0;
        } else {
                this.fps++;
        }
        var gl = this.gl;
        gl.bindFramebuffer(gl.FRAMEBUFFER, this.framebuffers.step);
        gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D,
             this.textures.back, 0);
        gl.bindTexture(gl.TEXTURE_2D, this.textures.front);
        gl.viewport(0, 0, this.statesize.x, this.statesize.y);
        resize(gl.canvas);
        this.programs.sand.use()
                .attrib('quad', this.buffers.quad, 2)
                .uniform('state', 0, true)
                .uniform('matrix1', vec3(1,0,0))
                .uniform('matrix2', vec3(0,1,0))
                .uniform('matrix3', vec3(0,0,1))
                .uniform('scale', this.statesize)
                .uniform('res', this.res)
                .draw(gl.TRIANGLE_STRIP, 4);
        this.swap();
        return this;
};

SAND.prototype.translation = function(tx, ty) {
        return [1, 0, 0, 0, 1, 0, tx, ty, 1,];
};

SAND.prototype.draw = function() {
        var gl = this.gl;
        gl.bindFramebuffer(gl.FRAMEBUFFER, null);
        gl.bindTexture(gl.TEXTURE_2D, this.textures.front);

        var z = 0;
        var mat = this.translation(z,z);
        var matrix1 = vec3(mat[0], mat[1], mat[2]);
        var matrix2 = vec3(mat[3], mat[4], mat[5]);
        var matrix3 = vec3(mat[6], mat[7], mat[8]);

        resize(gl.canvas);
        gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
        this.programs.draw.use()
                .attrib('quad', this.buffers.quad, 2)
                .uniform('matrix1', matrix1)
                .uniform('matrix2', matrix2)
                .uniform('matrix3', matrix3)
                .uniform('state', 0, true)
                .uniform('scale', this.viewsize)
                .uniform('shift', this.shift)
```

```
            .uniform('color', this.color)
            .draw(gl.TRIANGLE_STRIP, 4);
    return this;
};

SAND.prototype.texture = function() {
    var state = new Float32Array(this.statesize.x * this.statesize.y * 4);
    for (var i = 0; i < state.length; i += 1) {
        state[i] = 0;
    }
    var gl = this.gl;
    var tex = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, tex);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, this.statesize.x, this.statesize.y, 0, gl.RGBA,
        gl.FLOAT, state);
    return tex;
};

SAND.prototype.get = function() {
    var gl = this.gl, w = this.statesize.x, h = this.statesize.y;
    gl.bindFramebuffer(gl.FRAMEBUFFER, this.framebuffers.step);
    gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D,
        this.textures.front, 0);
    var state = new Float32Array(w * h * 4);
    gl.readPixels(0, 0, w, h, gl.RGBA, gl.FLOAT, state);
    for (var i = 0; i < state.length; i++) {
        state[i] = state[i]*max;
    }
    return state;
};

SAND.prototype.set = function(state) {
    var gl = this.gl;
    var rgba = new Float32Array(this.statesize.x * this.statesize.y * 4);
    for (var i = 0; i < state.length; i+=4) {
        rgba[i + 0] = state[i]/max;
        rgba[i + 1] = state[i + 1]/max;
        rgba[i + 2] = state[i + 2]/max;
        rgba[i + 3] = state[i + 3]/max;
    }

    gl.bindTexture(gl.TEXTURE_2D, this.textures.front);
    gl.texSubImage2D(gl.TEXTURE_2D, 0, 0, 0, this.statesize.x, this.statesize.y, gl.RGBA,
        gl.FLOAT, rgba);
    return this;
};

// this is what gets it running

SAND.prototype.start = function(n,m) {
    if (this.timer == null) {
        this.timer = setInterval(function(){
            for(var i = 0; i < n; i++){
                sand.step();
                }
            sand.draw();
        }, m);
    }
    return this;
};

SAND.prototype.stop = function() {
    clearInterval(this.timer);
    this.timer = null;
```

```
        return this;
};

SAND.prototype.toggle = function() {
        if (this.timer == null) {
                        this.start(this.speed, this.frames);
        } else {
                        this.stop();
        }
};

SAND.prototype.set_speed = function(n,m) {
        this.stop();
        this.start(n,m);
};

SAND.prototype.run = function(n) {
        for (var i = 0; i < n; i++){
                sand.step();
        }
        return this;
};

SAND.prototype.setRandom = function(p) {
        var gl = this.gl, size = this.statesize.x * this.statesize.y;
        var state = this.get();
        for (var i = 0; i <= size*4; i = i + 4) {
                var r = Math.random();
                for (var j = 1; j <= 4 ; j++){
                        if (r <= (j/4)){
                                state[i] = j - 1;
                                break;
                        }
                }
        }
        this.set(state);
};

SAND.prototype.set_surface = function(n) {
        var gl = this.gl, w = this.statesize.x, h = this.statesize.y;
        var state = this.get();

        switch(n){
                case 0:
                        for (var i = 0; i < state.length; i += 4) {

                                if (i % 3 == 0 || i % 5 == 0){
                                        state[i + 1] = 0;
                                }
                        }
                        break;

                case 1:
                        for (var i = 0; i < w; i++) {
                                for (var j = 0; j < h; j++) {

                                        if (i < (w - this.res.x)/2.0 || i > w - .5 - (w - this.res.x)/2.0
                                            || j < (h - this.res.y)/2.0 || j > h - .5 - (h -
                                            this.res.y)/2.0){

                                                state[(i + j*w)*4 + 1] = 0;
                                        } else {
                                                state[(i + j*w)*4 + 1] = 1;
                                        }
                                }
                        }
                        break;

```

```
			case 2:
				for (var i = 0; i < w; i++) {
					for (var j = 0; j < h; j++) {

						if ((i - w*.5)*(i - w*.5) + (j - h*.5)*(j - h*.5) > 1000.0) {

							state[(i + j*w)*4 + 1] = 0;
						} else {
							state[(i + j*w)*4 + 1] = 1;
						}
					}
				}
				break;

			case 3:
				for (var i = 0; i < w; i++) {
					for (var j = 0; j < h; j++) {

						if (j > 100.0 || j < 200.0 || i > 200.0 || i < 100.00){

							state[(i + j*w)*4 + 1] = 1;
						} else {
							state[(i + j*w)*4 + 1] = 0;
						}
					}
				}
				break;
		}
		this.set(state);
};

SAND.prototype.get_region = function(state) {
		var region = [];

		for (var i = 0; i < state.length; i += 4){
			if (state[i + 1] == 1){
				region.push(i);
			}
		}

		return region;
};

SAND.prototype.add_random = function(state) {
		var region = this.get_region(state);

		var r = Math.floor(Math.random() * region.length);
		state[region[r]] += 1;

		return state;
};

SAND.prototype.fullstate = function(n) {
		var state = this.get();
		for (var i = 0; i < state.length; i += 1){
			state[4*i] = n;
		}
		return state;
};

SAND.prototype.reset = function() {
		var gl = this.gl;
		var state = this.get();

		for (var i = 0; i < state.length; i += 1) {
			state[i] = 0;
		}
  }
```

```
  this.set(state);
        this.set_surface(this.shape_choice);
};

SAND.prototype.clear_firing_history = function() {
        var gl = this.gl;
        var state = this.get();

        for (var i = 0; i < state.length; i += 4) {
                state[i + 3] = 0;
  }

  this.set(state);
};

SAND.prototype.save = function() {
        this.saves.push(sand.get());
        this.save_id = this.save_id + 1;
};

SAND.prototype.load = function(n) {
        this.set(this.saves[n]);
};

SAND.prototype.brush = function(x, y, choice, type) {
  var gl = this.gl, w = this.statesize.x, h = this.statesize.y;
        var state = this.get();

        switch(type){
                case 0:
                        if (choice){
                                state[(x + y*w)*4] += 1;
                        } else {
                                state[(x + y*w)*4] -= 1;
                        }
                        this.set(state);
                        break;

                case 1:
                        if (choice){
                                state[(x + y*w)*4 + 1] = 0;
                        } else {
                                state[(x + y*w)*4 + 1] = 1;
                        }
                        this.set(state);
                        break;

                case 2:
                        if (choice){
                                state[(x + y*w)*4 + 1] = 2;
                        } else {
                                state[(x + y*w)*4 + 1] = 1;
                        }
                        this.set(state);
                        break;

                case 3:
                        if (choice){
                                state[(x + y*w)*4 + 1] = 3;
                        } else {
                                state[(x + y*w)*4 + 1] = 1;
                        }
                        this.set(state);
                        break;

                case 4:
                        if (choice){
                                state[(x + y*w)*4] = this.brush_height;
```

```
                    }
                    this.set(state);
                    break;

            case 5:
                    if (choice){
                            state[(x + y*w)*4] -= 4;

                            state[(x + 1 + y*w)*4] += 1;
                            state[(x - 1 + y*w)*4] += 1;
                            state[(x + (y + 1)*w)*4] += 1;
                            state[(x + (y - 1)*w)*4] += 1;
                    } else {
                            state[(x + y*w)*4] += 4;

                            state[(x + 1 + y*w)*4] -= 1;
                            state[(x - 1 + y*w)*4] -= 1;
                            state[(x + (y + 1)*w)*4] -= 1;
                            state[(x + (y - 1)*w)*4] -= 1;
                    }
                    state[(x + y*w)*4 + 2] = 10;
                    this.set(state);
                    break;

            case 6:
                    $("#inspect_val").val(state.slice((x+y*w)*4, (x+y*w)*4 + 4));
                    break;
        }
};

//called when clicking to add or delete cells from the region
SAND.prototype.draw_surface = function(x, y, choice){
        var gl = this.gl, w = this.statesize.x, h = this.statesize.y;
        var state = this.get();

        if (choice){
                state[(x + y*w)*4 + 1] = 1;
        } else {
                state[(x + y*w)*4 + 1] = 0;
        }

    this.set(state);
};

//calculates closeness of two states
SAND.prototype.distance = function(state_1, state_2){
        var d = 0;

        for (var i = 0; i < state_1.length; i = i + 4) {
                d += Math.pow(state_2[i] - state_1[i], 2);
        }

        return d;
};

SAND.prototype.markov_approximation = function(target) {
        var gl = this.gl, w = this.statesize.x, h = this.statesize.y;
        var init_state = this.get();

        //compare with target
        var d1 = sand.distance(init_state, target);

        //add a random grain
        var new_state = this.get();
        this.set(this.add_random(new_state));


        this.stabilize();
```

```

        //compare with target
        var d2 = sand.distance(new_state, target);

        //if further, return to initial state
        if (d2 > d1) {
                this.set(init_state);
        }

        //display the state
        sand.draw();

        return sand.distance(this.get(), target);
};

SAND.prototype.start_markov_approximation = function(target, n) {
        sand.toggle();
        if (this.markov_timer == null) {
                this.markov_timer = setInterval(function(){
                        for (var i = 0; i < n; i++) {
                                if (sand.markov_approximation(target) == 0){
                                        sand.pause_markov_approximation();
                                }
                        }
                }, 1);
  }
        sand.toggle();
};

SAND.prototype.pause_markov_approximation = function() {
        clearInterval(this.markov_timer);
        this.markov_timer = null;
};

// this function and the one below are what implement the ‘‘surface’’ method discussed in the paper
SAND.prototype.approx_identity_alg = function(n){
        //use approx_identity_4(n) to get close
        //fire sink until nothing changes

        v = this.approx_identity_4(n);
        this.fire_vector(v);

        //predict additional needed firings
        var k = 0.012857968994995O6*n*n + -0.14120481213637398*n + 3.916531993030239;

        this.fire_sink(k);
        this.stabilize(); // this takes time
        this.draw();
        this.fire_sink_until_id(); // this too
        this.draw();
};

SAND.prototype.approx_identity_4 = function(n) {
        //first guess coefficients

        var h  = Math.round(0.1674411791810444*n*n + 0.18971510117164725*n - 2.797811919063292);
        var c  = Math.round(-0.8361720629239193 + 1.4848313882485358*Math.log(n));
        var s  = Math.round(0.791548224489514*n - 1.158817405099287);

  var l = (n - 1)/2;
        var model = function(x, y) {return h + (s-h)*(x*x + y*y) + (c + h - 2*s)*((x*x)*(y*y));};

        //center and scale poly
        var p = function(x, y) {return -Math.round(model((x - l)/l, (y - l)/l));};

        //construct firing vector
        var v = new Float32Array(n*n);
        for (var j = 0; j < n; j++){
```

```
        for (var i = 0; i < n; i++){
            v[n*j + i] = p(i, j);
        }
    }
    //console.log(v);
    return v;
};

SAND.prototype.plus = function(n) {
    var state = sand.get();
    for (var i = 0; i <= state.length; i = i + 4){
        if (state[i + 1] == 1){
            for (var j = 0; j < n; j++){
                    state[i] = state[i] + 1;
                }
            }
        //}
    }
    sand.set(state);
};

SAND.prototype.minus = function(n) {
    var state = sand.get();
    for (var i = 0; i <= state.length; i = i + 4){
        if (state[i] - n >= 0) {
            state[i] = state[i] - n;
        } else {
            state[i] = 0
        }
    }
    sand.set(state);
};

SAND.prototype.dualize = function() {
    var state = sand.get();
    for (var i = 0; i <= state.length; i += 4){
        state[i] = 3 - state[i];
    }
    sand.set(state);
};

SAND.prototype.check_stable = function() {
    var gl = this.gl, w = this.statesize.x, h = this.statesize.y;
  var state = this.get();

    for (var i = 0; i < w * h * 4; i = i + 4) {
        if (state[i + 2] == 10 || state[i + 2] == 11){
            return 1;
        }
    }

    return 0;
};

SAND.prototype.stabilize = function() {
    var gl = this.gl, w = this.statesize.x, h = this.statesize.y;
    var state = this.get();

    this.step();

    sand.set_speed(100,1);
    for (var i = 0; i < w * h * 4; i = i + 4) {
        if (state[i + 1] == 2){
            alert("Cannot stabilize when source cells are present.");
            return 0;
        }
    }

```

```
        // this seems really sensitive in total time elapsed to the choice of maximum i here,
            investigate further
        while (this.check_stable()){
              for(var i = 0; i < 10000; i++){
                    this.step();
              }
        }

        sand.set_speed(1,1);
        this.draw();
        return 1;
};

SAND.prototype.set_identity = function() {
        // deprecated with introduction of approximate_identity_alg
        alert("This may take a while.");
        this.reset();
        this.fire_sink(this.approx_k());
        this.fire_sink_until_id([0, 0, 1000, 1, 1]);
        this.identity = sand.get();
};

SAND.prototype.rec_inverse = function() {
        this.toggle();
        this.plus(6);
        this.stabilize();
        this.dualize();
        this.plus(3);
        this.stabilize();
        this.toggle();
        this.draw();
};

//this function reads a state array and creates a firing vector out of the firing history
SAND.prototype.get_firing_vector = function(state){
        var region = this.get_region(state);

        var vector = new Float32Array(region.length);
        for (var i = 0; i < vector.length; i += 1){
              vector[i] = state[region[i] + 3];
        }
        return vector;
};

SAND.prototype.save_firing_vector = function(){
        var gl = this.gl, w = this.statesize.x, h = this.statesize.y;
    var state = this.get();

        this.firing_vectors.push(sand.get_firing_vector(state));
        this.firing_vector_id = this.firing_vector_id + 1;
};

SAND.prototype.fire_vector = function(vector) {
        var gl = this.gl, w = this.statesize.x, h = this.statesize.y;

        var state = this.get();
        var region = this.get_region(state);
        var newstate = this.get();

        for (var i = 0; i < vector.length; i += 1){
              var j = region[i];
              var n = vector[i];
              newstate[region[i]] -= 4*n;

              newstate[j + 4] += n;
              newstate[j - 4] += n;
              newstate[j + 4*w] += n;
              newstate[j - 4*w] += n;
```

```
                newstate[j + 3] += n;
        }

        sand.set(newstate);
        sand.draw();
        return 1;
};

SAND.prototype.set_max_inverse = function(){
        sand.stop();
        sand.reset();
        sand.set_identity();
        this.cmax_inverse_vector = sand.get_firing_vector(sand.identity);
        return 1;
};

SAND.prototype.add = function(state1, state2) {
        //note that the allowed region comes from state1
        var state = new Float32Array(state1.length);

        for (var i = 0; i <= state1.length; i += 4){
                if (state1[i + 1] == 1){
                        state[i] = state1[i] + state2[i];
                        state[i + 1] = 1;
                } else {
                        state[i + 1] = 0;
                }

        }
        return state;
};

SAND.prototype.eventCoord = function(event) {
        var $target = $(event.target),
        offset = $target.offset(),
        border = 1,
        x = event.pageX - offset.left - border,
        y = $target.height() - (event.pageY - offset.top - border);
        return vec2(Math.floor((x + this.shift.x) / (this.scale)), Math.floor((y + this.shift.y) /
            this.scale));
};

SAND.prototype.fire_sink = function(n){
        var state = this.get();
        var region = this.get_region(state);
        var vector = new Float32Array(region.length);

        for (var i = 0; i < vector.length; i += 1){
                vector[i] = -n;
        }

        this.fire_vector(vector);
};

SAND.prototype.is_equal = function(state1, state2){
        for (var i = 0; i < state1.length; i += 4){
                if (state1[i] != state2[i]){
                        return 0;
                }
        }
        return 1;
};

// fires sink until hits identity
SAND.prototype.fire_sink_until_id = function(){

        // being weirdly slow
```

```
        var newstate, oldstate;
        var counter = 0;
        var equal = 0;

        while(!equal){

                oldstate = this.get();

                this.fire_sink(1);
                this.stabilize();

                newstate = this.get();

                if (!this.is_equal(newstate, oldstate)){
                        counter += 1;
                } else {
                        equal = 1;
                        this.set(oldstate);
                }
        }
};

SAND.prototype.approx_k = function() {
        return Math.floor((2/3)*(Math.floor(sand.m/2)*Math.floor(sand.m/2)) +
             .40476*(Math.floor(sand.m/2)) + .40476/2)
};

SAND.prototype.time_burning_config_method = function() {
        k = this.approx_k();
        sand.reset();
        var t0 = performance.now();
        this.fire_sink(k)
        this.fire_sink_until_id();
        var t1 = performance.now();
        alert("Calculation took " + (t1 - t0) + " milliseconds.")
};

// all these approx_identities are deprecated except for approx_identity_4, but I'm keeping them here
     for now

SAND.prototype.approx_identity = function(n) {
        //first guess coefficients
        var coeffs = this.approx_coeffs(n);
        var h = coeffs[0]
        var c = coeffs[1]
        var s = coeffs[2]

        //create firing vector
        var v = this.approx_firing_vector(n, h, c, s, 0);
        return v;
};

SAND.prototype.approx_identity_2 = function(n) {
        //first guess coefficients
        var h = -0.16573652165412933*n*n + -0.7710039875902805*n + -0.5866930171310152
        var c = 0.0014357061858030207*n*n + -0.13699963669877713*n + -1.4496706192412137
        var s = -0.0004727325274926919*n*n + -0.7596584069827825*n + -0.7816864295162682

        var l = (n - 1)/2
        var model = function(x, y) {return h + (s-h)*(x*x + y*y) + (c + h - 2*s)*((x*x)*(y*y));};

        //center and scale poly
        var p = function(x, y) {return Math.round(model((x - l)/l, (y - l)/l));};

        //construct firing vector
        var v = new Float32Array(n*n);
        for (var j = 0; j < n; j++){
```

```
            for (var i = 0; i < n; i++){
                v[n*j + i] = p(i, j);
            }
        }
        return v;
};

SAND.prototype.approx_firing_vector = function(n, h, c, s, d) {
        //alert([n,h,c,s,d])
        var l = (n - 1)/2
        var model = function(x, y) {return h + (s-h)*(x*x + y*y) + (c + h - 2*s - 2*d)*((x*x)*(y*y))
            + d*((x*x)*(y*y*y*y) + (x*x*x*x)*(y*y));};

        //center and scale poly
        var p = function(x, y) {return Math.round(model((x - l)/l, (y - l)/l));};

        //construct firing vector
        var v = new Float32Array(n*n);
        for (var j = 0; j < n; j++){
            for (var i = 0; i < n; i++){
                v[n*j + i] = p(i, j);
            }
        }
        return v;
};

SAND.prototype.approx_coeffs = function(n){
        var h = -0.16573652165412933*n*n + -0.7710039875902805*n + -0.5866930171310152
        var c = 0.0014357061858030207*n*n + -0.13699963669877713*n + -1.4496706192412137
        var s = -0.0004727325274926919*n*n + -0.7596584069827825*n + -0.7816864295162682
        return [h, c, s];
};


SAND.prototype.approx_identity_3 = function(n, d) {
        //first guess coefficients

        var coeffs = this.approx_coeffs(n);
        var h = coeffs[0]
        var c = coeffs[1]
        var s = coeffs[2]

        /* var h = -0.16573652165412933*n*n + -0.7710039875902805*n + -0.5866930171310152
    var c = 0.0014357061858030207*n*n + -0.13699963669877713*n + -1.4496706192412137
    var s = -0.0004727325274926919*n*n + -0.7596584069827825*n + -0.7816864295162682
 */
    var l = (n - 1)/2
        var model = function(x, y) {return h + (s-h)*(x*x + y*y) + (c + h - 2*s - 2*d)*((x*x)*(y*y))
            + d*((x*x)*(y*y*y*y) + (x*x*x*x)*(y*y));};

        //center and scale poly
        var p = function(x, y) {return Math.round(model((x - l)/l, (y - l)/l));};

        //construct firing vector
        var v = new Float32Array(n*n);
        for (var j = 0; j < n; j++){
            for (var i = 0; i < n; i++){
                v[n*j + i] = p(i, j);
            }
        }
        //console.log(v);
        return v;
};

SAND.prototype.zoom = function(dz, n) {
        if (n < 0) {
            if (sand.viewsize.x - dz >= 300){
                sand.viewsize.x -= dz;
```

```
                        sand.viewsize.y -= dz;
                        sand.shift.x -= dz/2;
                        sand.shift.y -= dz/2;
                }
        } else {
                sand.viewsize.x += dz;
                sand.viewsize.y += dz;
                sand.shift.x += dz/2;
                sand.shift.y += dz/2;

        }
        sand.draw();
};

// this function listens for mouse inputs and some keyboard inputs
function Controller(SAND) {
        this.sand = sand;
        var _this = this,
                        $canvas = $(sand.gl.canvas);
        this.drag = null;
        $canvas.on('mousedown', function(event) {
                if (sand.brush_type == 7){
                        _this.drag = event.which;
                        var mx = event.clientX;
                        var my = event.clientY;
                } else {
                        event.preventDefault();
                        _this.drag = event.which;
                        var pos = sand.eventCoord(event);
                        sand.brush(pos.x, pos.y, _this.drag == 1, sand.brush_type);
                        sand.draw();
                }
        });

        $canvas.on('mouseup', function(event) {
                _this.drag = null;
        });

        $canvas.on('mousemove', function(event) {
                if (sand.brush_type == 7){
                        event.preventDefault();
                        if (_this.drag) {
                                var mx = event.clientX;
                                var my = event.clientY;

                                console.log('Mouse position: ' + mx + ',' + my);
                                console.log('View shift: ' + sand.shift.x + ',' + sand.shift.y );

                                sand.shift.y = Math.max(my - sand.shift.y, my);
                                sand.draw();
                        }
                } else {
                        event.preventDefault();
                        if (_this.drag) {
                                var pos = sand.eventCoord(event);
                                sand.brush(pos.x, pos.y, _this.drag == 1, sand.brush_type);
                                sand.draw();
                        }
                }

        });

        $canvas.on('contextmenu', function(event) {
                        event.preventDefault();
                        return false;
        });

        // copied and modified from some jsfiddle that I can't find again
```

```
        $('#sand').bind('mousewheel DOMMouseScroll', function(e) {
                var scrollTo = 0;
                e.preventDefault();
                if (e.type == 'mousewheel') {
                        scrollTo = (e.originalEvent.wheelDelta * -1);
                        sand.zoom(sand.dz, -e.originalEvent.wheelDelta);
                }
                else if (e.type == 'DOMMouseScroll') {
                        scrollTo = 40 * e.originalEvent.detail;
                        sand.zoom(sand.dz, -e.originalEvent.detail);
                }
                $(this).scrollTop(scrollTo + $(this).scrollTop());
        });

        $(document).on('keyup', function(event) {
                        switch (event.which) {

                case 46: /* [delete] */
                        sand.reset();
                        sand.draw();
                        break;
                case 32: /* [space] */
                        sand.toggle();
                        break;
                case 87:
                        // up
                        sand.shift.y += sand.dx;
                        sand.draw();
                        break;
                case 83:
                        //down
                        sand.shift.y -= sand.dx;
                        sand.draw();
                        break;
                case 65:
                        //left
                        sand.shift.x -= sand.dx;
                        sand.draw();
                        break;
                case 68:
                        //right
                        sand.shift.x += sand.dx;
                        sand.draw();
                        break;
                case 109:
                        //-
                        sand.zoom(sand.dz, -1);
                        break;
                case 107:
                        //+
                        sand.zoom(sand.dz, 1);
                        break;
                }
        });
}

$(window).on('keydown', function(event) {
    return !(event.keyCode === 32);
});

function download(data, name) {
  var link = document.createElement("a");
  link.download = name;
  var uri = data;
  link.href = uri;
  document.body.appendChild(link);
  link.click();
  document.body.removeChild(link);
```

```
  delete link;
}

function copyToClipboard(text) {
  window.prompt("Copy to clipboard: Ctrl+C, Enter", text);
}

// initialize the sandpile on the canvas
var sand = null, controller = null;
$(document).ready(function() {
    var $canvas = $('#sand');
    sand = new SAND($canvas[0], 8).draw().start(1, 1);
    controller = new Controller(sand);
});
```

Further Reading

Angel, E., & Shreiner, D. (2015). *Interactive Computer Graphics: a top-down approach with WebGL*. Boston, MA: Pearson.

Bak, P., Tang, C., & Wiesenfeld, K. (1987). Self-organized criticality - An explanation of 1/f noise. *Physical Review Letters*, *59*, 381–384.

Caracciolo, S., Paoletti, G., & Sportiello, A. (2008). Explicit characterization of the identity configuration in an abelian sandpile model. *Journal of Physics A: Mathematical and Theoretical*, *41*(49).

Dhar, D. (1992). The abelian sandpile model of self-organized criticality. *AIP Conference Proceedings*.

Levine, L. T. (2007). *Limit theorems for internal aggregation models*. ProQuest LLC, Ann Arbor, MI. Thesis (Ph.D.)–University of California, Berkeley. `http://gateway.proquest.com/openurl?url_ver=Z39.88-2004&rft_val_fmt=info:ofi/fmt:kev:mtx:dissertation&res_dat=xri:pqdiss&rft_dat=xri:pqdiss:3306223`

Pegden, W., & Smart, C. K. (2013). Convergence of the Abelian sandpile. *Duke Math. J.*, *162*(4), 627–642. `http://dx.doi.org/10.1215/00127094-2079677`

Perkinson, D., & Corry, S. (2016). *Divisors and Sandpiles*. `http://people.reed.edu/~davidp/divisors_and_sandpiles/draft-11.20.2016.pdf`

Wellons, C. (2014). null program. `http://nullprogram.com/blog/2014/06/10/`