Implementation of a Lattice Gas Simulation

Tony H. Kim

Spring 2007

1 Introduction

In this paper I discuss the implementation of a lattice-gas simulation on a hexagonal grid. The application is written in the C++ language using the Windows and DirectX APIs. Due to the nature of the project, following the initial introduction to the lattice gas concept, the discussion has somewhat the feel of a software manual. After documenting my application in Section 3, I spend some time discussing the technical difficulties associated with attempting to simulate a large number of (albeit simple) automata; and the strategies employed in order to reduce the cost of computations. I conclude with a gallery consisting of screens from the various demos I have produced using the engine.

2 The Lattice Gas

2.1 A Simple Description

The lattice gas particle is an entity that hops from point to point on the discrete sites of a lattice with each time step (Figure 1). These spatial and temporal discretizations imposed on the lattice gas particle make the model a highly artificial system to begin with, but there are additional simplifications that we further assume. For instance:

- Owing to the hexagonal grid, the particle can move in only one of six directions.

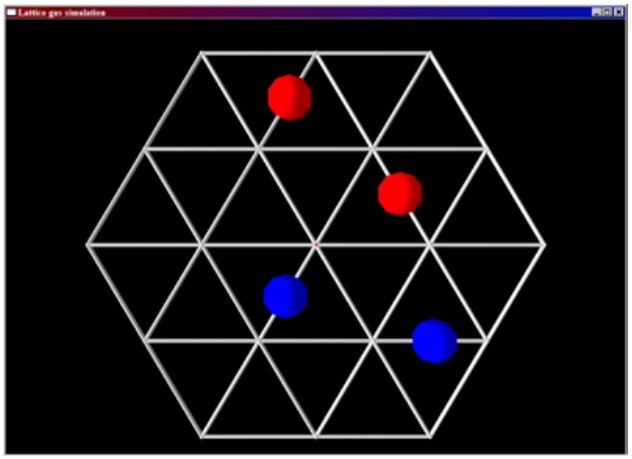


Figure 1: The lattice gas system as visualized in my application

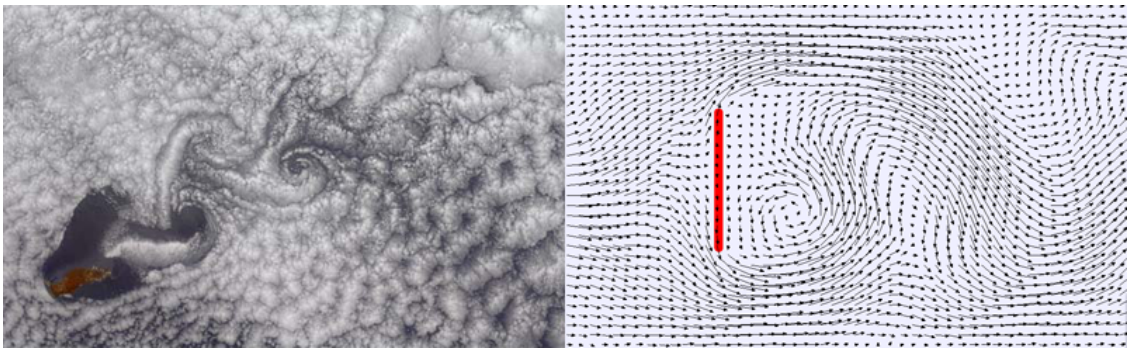


Figure 2: At large scales, the lattice gas produces the qualitative features of real flows

- There cannot exist more than one particle at each node heading in the same direction. This implies that each node can support up to a maximum of six particles at a particular instant.[1]
- The magnitude of the particle velocity is fixed.
- A particle cannot remain stationary. (Note: This is relaxed in more complicated implementations which may explicitly allow for stationary particles. However, the restrictions presented here correspond to the version I have produced.)

Given these severe, unphysical assumptions of the lattice gas model, it is questionable whether we are actually "simulating" anything – let alone the complex interactions and behaviors of real gases and fluids. Given the alarmingly simple framework, it is remarkable that at large enough scales the lattice gas reproduces some of the features associated with real flows. A well-known example of this is shown above in the lattice gas reproduction of the von Karman vortex street phenomenon (Figure 2). This and other surprising successes have led some to characterize the lattice gas concept as one that "has nearly nothing in common with real fluids except for one special property – at a macroscopic scale it flows just like them!"[4]

2.2 Advantages of the lattice gas

If our goal is to attempt the simulation of physical particles, one may reasonably ask: given the performance of modern computers and special-purpose machines, why not simply create a more realistic application that models the interactions of particles in continuous space, and continuous time?[2] While straightforward, the computational order of growth associated with such an approach prevents simulations at a size large enough to be interesting.

Intuitively, the problem scales as $O(N^2)$, where N denotes the number of particles being simulated. At every frame, for each of the N particles, we must query $N-1$ others for a possible collision. This is an underestimate, however, for we must also take into account three-particle collisions, reflections off of walls, and other auxiliary calculations. Of course, we have not yet even considered the collision computation itself, which, for just two spherical particles in the plane, will involve many parameters such as the angle of incidence or even the size of the particles, in addition to the initial velocities.

Hence, to feasibly simulate a large number of particles, we must be willing to make simplifications to the physical model. The lattice gas represents one extreme set of such simplifications. In particular, it is of interest because of its unusual $O(n)$ order of growth in the computational effort. As we will find later, this owes to the fact that the relevant parameter for the "size" of the lattice gas simulation is *not* the number of particles present, but rather the number of nodes; and that these nodes, in turn, have (more-or-less) *fixed*

[1] Physics joke: the lattice gas particle must have half-integer spin.
[2] As "continuous" as they can be, implemented in a computer program.

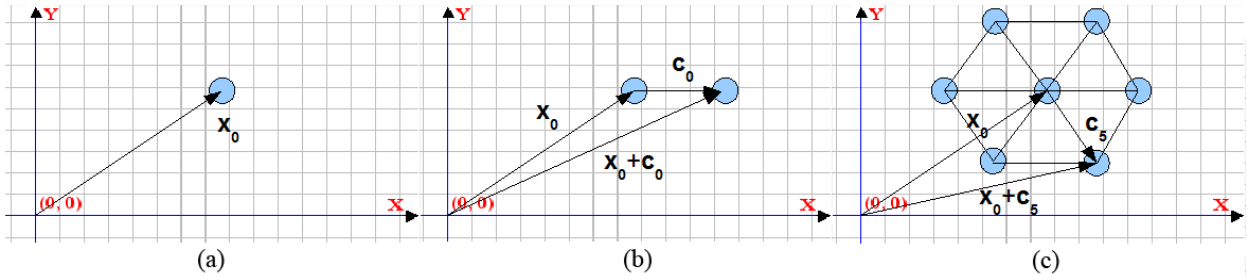


Figure 3: Creating nodes in the coordinate plane

neighbors. We will return to this point in Section 2.7, after discussing the means to describe analytically the lattice gas system.

2.3 The analytic description of the lattice network

In order to construct the lattice, we begin with a coordinate plane. Consider a node at the vector position $\vec{x}_0$ as in Figure 3(a). For simplicity, we require that there exist only one node per location on the plane. We may then use the vector $\vec{x}_0$ to refer to the *node* at the location $\vec{x}_0$. This is convenient notation, and I will be utilizing it throughout the discussion.

Once we have a single node at $\vec{x}_0$, we may construct the rest of the lattice *with respect to* $\vec{x}_0$ with the help of "displacement" vectors $\vec{c}_i$ whose directions in the plane are given by:

$$\vec{c}_i = (cos(\pi i/3), sin(\pi i/3)), \text{ where } i = 0, 1, 2, 3, 4, 5;$$

In addition, the magnitude of $\vec{c}_i$ denotes the desired spacing between adjacent nodes of the lattice. Using this construct, we may place a node at $\vec{x}_0 + \vec{c}_0$ as in Figure 3(b).

Having repeated the above process for each direction i, we expect the node $\vec{x}_0$ to be connected to the six nodes at locations $\vec{x}_0 + \vec{c}_i$. (See Figure 3(c)) This is so because the connections are made based on the relative positions of the nodes on the coordinate plane.

While the discussion above may be trivial, it is still of some importance for my program, because the process of node generation described here is actualy quite versatile; and it is *exactly* the same procedure that my program follows. For more information on how to procedurally generate nodes in the application, please consult the files NodeGeometry.h/.cpp.

2.4 Denoting particles on the lattice

The restriction of one particle per one direction of a site gives rise to a convenient notation for denoting the occupancy of a node. Let $\vec{n}(\vec{x})$ be a six-component vector function of a node $\vec{x}$ whose components $n_i(\vec{x})$ denote the presence ($n_i = 1$) or absence ($n_i = 0$) of a particle heading in the i-th direction.

So, for the example of $\vec{x_0}$ in Figure 4, the occupation vector may be written:

$$\vec{n}(\vec{x}_0) = (n_0, n_1, n_2, n_3, n_4, n_5) = (1, 1, 0, 0, 1, 0) \tag{1}$$

Note that each of the entries n_i are "Boolean" variables in the sense that they possess only two possible values. The fact that such little information is sufficient to capture the complete state of the system owes to the remarkable simplicity of the lattice gas model. Later, we exploit this simplicity to achieve a very efficient method for collision calculations.

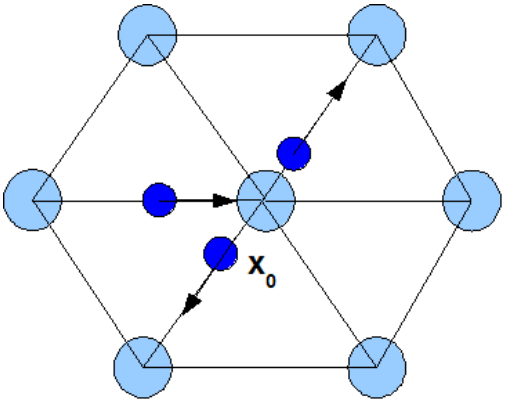


Figure 4: A possible particle occupation for the node $\vec{x}_0$

2.5 The "equation of motion" for the lattice gas

The state of the system is specified for some particular time if $\vec{n}$ is given for every node $\vec{x}_j$ present in the lattice. Given this information, how do we produce the subsequent state? The answer to this question is captured in the following equation:

$$n_i(\vec{x} + \vec{c}_i, t + 1) = n_i(\vec{x}, t) + \Delta_i[\vec{n}(\vec{x}, t)] \tag{2}$$

As defined previously, the first two terms denote the presence or absence of a particle heading in a particular direction i; they are evaluated at adjacent nodes $\vec{x}$ and $\vec{x} + \vec{c}_i$ and at successive times. The second term on the right Δ_i represents the "collision operator" that acts on the entire node occupation vector. The momentum operator can output a range of values -1,0,1 and hence can modify the "trajectory" of a particle.

To illustrate this point, consider the simple case below:

Suppose that at time t, the node at $\vec{x}_0$ posseses a sole particle heading in the $i = 1$ direction. Then we may ask whether there will be a particle at the node $\vec{x}_0 + \vec{c}_1$ at time $t + 1$ heading in the $i = 1$ direction. Physical intuition (Another reminder that we are in fact simulating a physical system!) would indicate that the particle would keep moving along its path; i.e. that $n_1(\vec{x}_0 + \vec{c}_1) = 1$. We can see how this result follows from the dynamical equation. For $i = 1$ we have from Eq. 2:

$$n_1(\vec{x} + \vec{c}_1, t + 1) = n_1(\vec{x}, t) + \Delta_1[\vec{n}(\vec{x}, t)] \tag{3}$$

Presumably, for a single particle, there will be no "collision" and the collision operator will yield $\Delta_1[\vec{n}(\vec{x}, t)] = 0$. This in turn gives:

$$\begin{aligned} n_1(\vec{x} + \vec{c}_1, t + 1) &= n_1(\vec{x}, t) \\ &= 1 \end{aligned}$$

In other words, the particle does in fact continue on its path.

In contrast, suppose now we have the following three-body heads-on collision scenario (Figure 5), i.e.:

$$\vec{n}(\vec{x}) = (1, 0, 1, 0, 1, 0) \tag{4}$$

In such a case, the collision operator will yield $\Delta_1[\vec{n}(\vec{x}, t)] = -1$, indicating that the particle does *not* continue on its original path. (Since now: $n_1(\vec{x}_0 + \vec{c}_1) = 0$) Simply put, the collision operator Δ_i is a mapping from

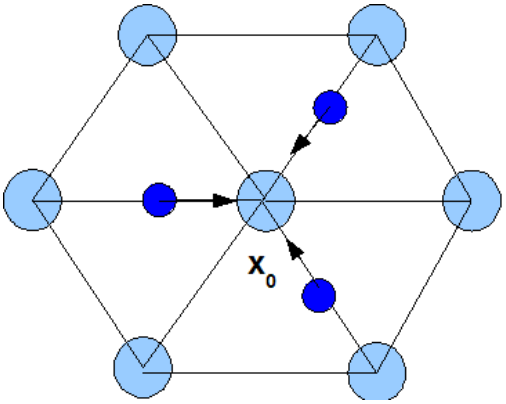


Figure 5: A possible particle occupation for the node $\vec{x}_0$

the set of occupation states at a node (all the possible values of $\vec{n}$) to the integers -1,0,1 that modify the trajectories of particles.

2.6 More on the collision operator

In the lattice gas approach, the collision operator Δ_i alone incorporates the possible ways in which particles interact with one another. The particular form of the collision operator chosen differs among implementations, and determines the complexity of each model. In my application, I employ the simplest collision operator, that handles two-body and three-body collisions.[3] This form of the collision operator is known as the FHP-I model (Frisch, Hasslacher, Pomeau).[4]

2.7 Remark on the computational order of growth

Consider for a moment a more physical simulation of N-particle dynamics based on Newton's laws of motion and gravitation. For that approach, the relevant equation of motion in two-dimensions is:

$$\ddot{\vec{x_j}} = \sum_{k=1}^{N} (Gm_k/(r_k - r_j)^2) \tag{5}$$

for the j-th particle.

An important feature to note is that, here, the relevant index j is over the *particles*. Hence, the number of particles being simulated is the relevant measure of the "size" of the simulation.

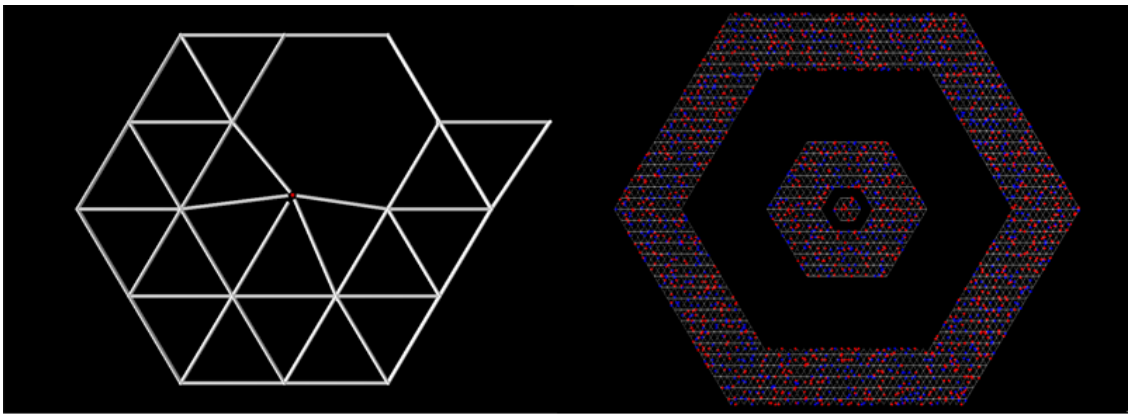
In contrast, for the lattice gas, the analogous equation reads:

$$n_i(\vec{x_j} + \vec{c}_i, t+1) = n_i(\vec{x_j}, t) + \Delta_i[\vec{n}(\vec{x_j}, t)] \tag{6}$$

which is to be computed over all *nodes* $\vec{x}_j$. Hence, the size parameter in this case is the number of *nodes* in the lattice. It is amusing to note that virtually the same number of computations will be needed to simulate a lattice of *no particles* as it would to simulate a completely saturated lattice!

Finally, note that in the case of the Newtonian simulation, the number of terms on the right-hand side increases proportionally to the size of the problem. However, in the case of the lattice gas, the computations

[3]There were slight modifications made to allow for two distinguishable types of particles.



(a) The node network is manipulated at runtime

(b) Simulation of large numbers of two distinctly colored particles.

on the RHS are fixed in number. Without detailed analysis, this is the main observation that leads us to conclude that the order of growth in the lattice gas simulation is $O(n)$, rather than $O(n^2)$.

3 Program documentation

3.1 Feature list

In developing the application, I have kept the following objectives and features in mind:

- Ability to generate and modify the lattice network at runtime. This allows for great control in creating customized "terrains" for the particles. (Figure 3.1(a))
- Ability to simulate at least $N > 1000$ particles at a reasonable speed.[4] (Figure 3.1(b))
- Simulate two distinctly colored particles. (Figure 3.1(b))
- Implementation of various behaviors at the boundaries.

3.2 Program control

Interaction with the software requires both the keyboard and the mouse. The commands are as follows.

Mouse:

- Left-click: Select (or unselect) a node;
- Right-click: Create a new node in the clicked location;
- Mouse-scroll: Zoom in/out;

Keyboard:

- W,S,A,D: Move the camera up/down/left/right respectively if no node is selected; If a node is selected, moves the location of the selected node.

[4]My benchmark is my own laptop - 1.6GHz, 256MB memory and a practically nonexistent graphics card!

- R,F: Zoom the camera in/out;
- Q: Delete the selected node;
- C: Clear all particles from the nodes;
- V: Randomly fill the node network with particles;
- 1,2,3,4,5,6,0: If a node is selected, then toggles a blue particle in the particular direction. ($i = 6 \Leftrightarrow i = 0$). If left-shift is held while the number is pressed, then will toggle a *red* particle;
- Space-bar: Unselect a node

3.3 Code organization

The diagram on the following page briefly describes the source files that comprise my application and their dependencies:

3.4 Further details on the development environment

The program was written using the C++ language of the Microsoft Visual Studio package, using the Windows and DirectX APIs. (So I'm quite confident the code will not run on a Mac or on Athena.) Furthermore, I have used a very recent release of the DirectX software development kit (SDK February 2007) so that the runtime software may need to be updated in order to run the demonstration executables.

In addition, on some machines I have encountered a "d3dx9_32.dll missing" error. This issue is discussed at and can be resolved by consulting:

```
http://www.toymaker.info/Games/html/d3dx_dlls.html
```

But I understand if you would not want to install additional software just to get my lattice simulations to run. (That's why I've included Section 5!)

4 Implementation details

4.1 Runtime generation of the node network

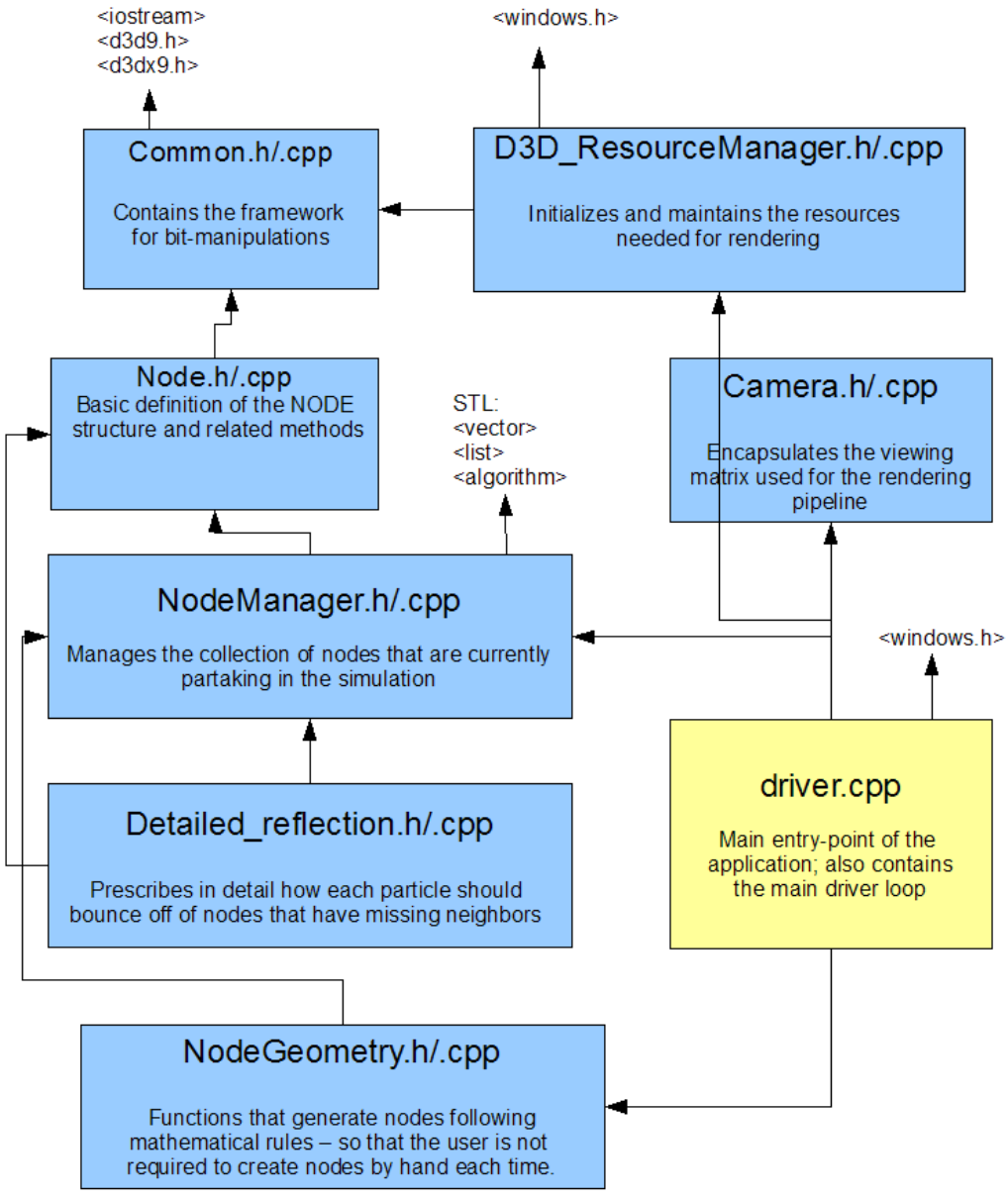
Having decided on runtime lattice generation, the pertinent programming challenge is: How does a node, acting independently, find its neighbors and make connections?

To resolve this issue, we must turn to the underlying coordinate system since that is the only framework that we may rely on initially. My solution relies on a data structure dubbed the "NodeManager" whose main purpose is to rapidly retrieve, for a given input coordinate (x_0, y_0), the node that is sufficiently closest to that target.

As seen in Figure 6, the NodeManager is comprised of two vector and list containers. The actual node data – which is represented as a C struct (in node.h) – is housed in the list. Throughout the execution, the node data stays relatively constant in memory. The primary operations that we perform on this list of "node-data" are addition and deletion, both of which are well suited for the list.[1]

The two vectors, on the other hand, contain pointers to the elements of node-data; and are sorted by the properties of the pointed data. One vector is sorted by the x-coordinate of the pointees; and the other by the y-coordinate. The two vectors are kept ordered at all times through careful insertions and modifications.

PROGRAM "ORGANIZATION"



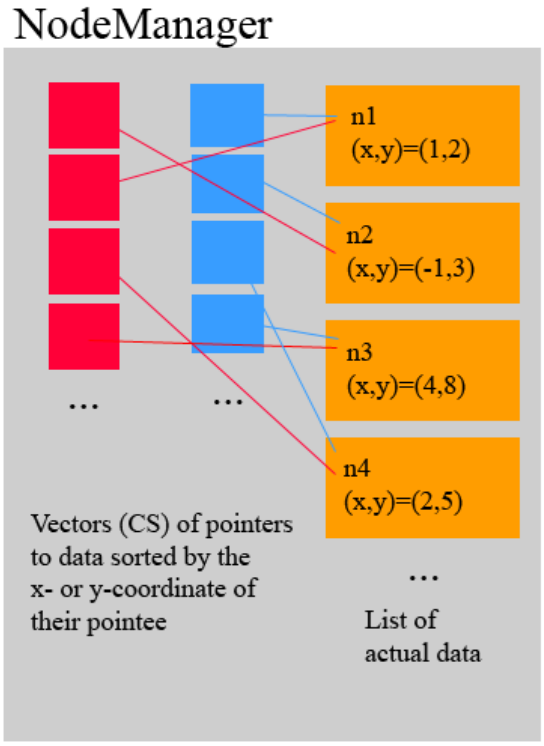


Figure 6: Representation of the "NodeManager" data structure

The purpose of this construct is the following. Suppose we have a collection of N nodes at arbirary points of the coordinate plane, and were interested in the node that is located at the point (x_0, y_0) within some tolerance. Without proper design, to answer this question, we would have to test all N nodes for their distances from the target point. The scaling properties of this solution isn't bad; however my solution is far superior.

The approach is the following:

1. For some tolerance t,
2. Determine the subset of the x-pointers whose pointees have a x-coordinate within the range $[x_0 - t, x_0 + t]$. This occurs in $O(log(n))$, since we take advantage of the binary search capabilites of the vector.
3. Determine the subset of the y-pointers whose pointees have a y-coordinate within the range $[y_0 - t, y_0 + t]$. This occurs in $O(log(n))$.
4. Find the intersection of the two resulting sets. This gives the pointers of all nodes whose x- and y-coordinates lie within the prescribed tolerance of the input target. While the intersection operation proceeds at $O(n)$, the previous subset operations have drastically reduced the size of the candidate nodes, so that we may consider this step as taking place in nearly constant time.[5]
5. Cycle through the elements of the intersection and find the node that is closest to the target. Again, since the size of this set is so small in comparison to the entire collection of nodes, we can consider this to scale at $O(1)$.

[5] I simply mean to say, that as long as the particle distribution is not becoming more *dense* with respect to the tolerance t, the results of the previous two operations should give subsets whose sizes are relatively independent of the actual number of nodes present in the lattice.

With this structure we can determine whether there are nodes at a particular coordinate location in effectively $O(log(n))$ time. This is ideal performance and allows my application to rely on the NodeManager extensively even for very large lattices. Hence, the runtime node network generation revolves around each node querying the NodeManager for nodes in their vicinity. (More specifically, at the locations $\vec{x} + \vec{c}_i$ where $\vec{x}$ denotes its own position.)

4.2 Rapid collision calculations

While an efficient algorithm for establishing node neighbors is helpful, it is in the overall operation of the program only a "minor" issue since it is invoked infrequently; e.g. it is called only when the network is first generated or modified. Each node, after establishing its neighbors, records the relevant identifiers so that it does not require the NodeManager apparatus for future computations.

On the other hand, for *every* frame we must perform the collision computations as described in Section 2.5. Hence, *this* is the process that needs to be optimized for maximum performance.

Previously I have remarked that the state of the lattice can be recorded in a sequence of boolean values $\vec{n}(\vec{x}_j)$. This translates naturally, in the C language, to representation using the low six-bits of a byte. This simply means to say that the occupation vector of each node will be encoded in binary. Hence, the occupation vectors $\vec{n}(\vec{x}) = (1, 1, 0, 0, 1, 0)$ and $\vec{n}(\vec{x}) = (1, 0, 1, 0, 1, 0)$ (encountered in Sections 2.4 and 2.5) would be encoded as: 00010011 and 00010101 respectively. In the convention I have adopted, the i-th component of $\vec{n}$ is stored in the i-th order bit of the byte.

Now recall that the collision operator (Section 2.5) is a mapping from various possible values of $\vec{n}$ to the set $-1, 0, 1$. Hence, in order to implement the collision operator, we must catch certain $\vec{n}$ configurations and return the appropriate integer. It is this checking for configurations that can be computed extremely efficiently in the bitwise representation. A brief example: How would one check, using the binary representation, whether some configuration represented a three-body heads-on collision?

Suppose we are given as the current state, the configuration as in Figure 4. We are interested in whether the scenario corresponds to a three-body heads on collision as in Figure 5. Within the program, the two are represented as $n_{current} = 00010011$ and $n_{three-body} = 00010101$, and it is already obvious that the two do not match. However, the computer verifies this in two steps:

1. First, the program computes the bitwise AND result of the two bytes, to produce: $n_{masked} = 00010001$;
2. Next, the program checks whether n_{masked} equals $n_{three-body}$. The equality of $n_{current}$ and $n_{three-body}$ corresponds exactly to the equality of n_{masked} and $n_{three-body}$.

The advantage of the bit representation is that the comparison of occupations can be made "at once" without checking the individual components. My implementation relies on this bitwise manipulations to rapidly conduct the collision computations.[6]

5 Gallery

I'll end with screenshots from the application.

[6]Strictly speaking, the masking stage isn't absolutely necessary, but since the representation does not use the information in the two highest bits of the byte, it is better to be on the safe side and to perform the masks each time to suppress whatever values that may linger in the two unused bits.

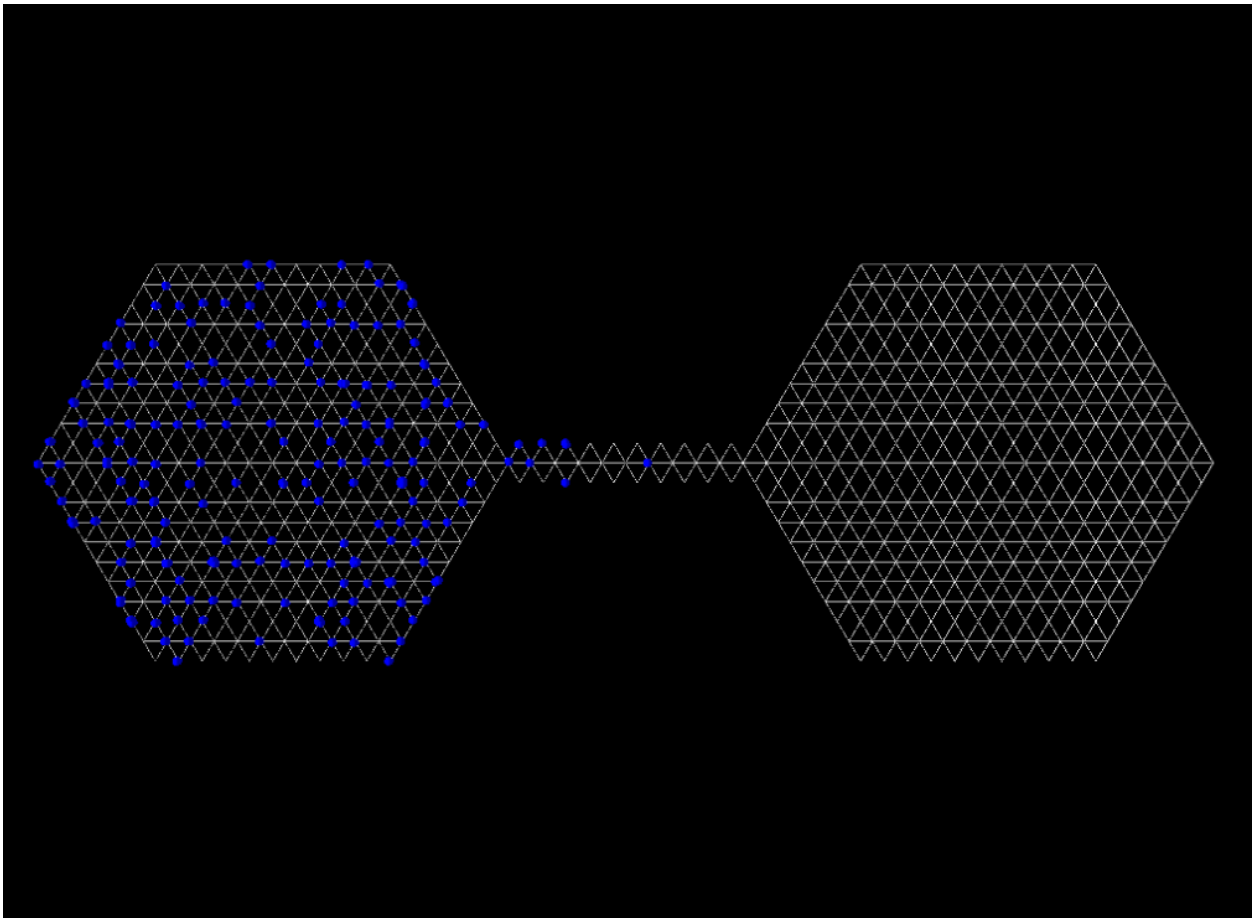


Figure 7: Experiment 1: The particles are initially concentrated in the left container.

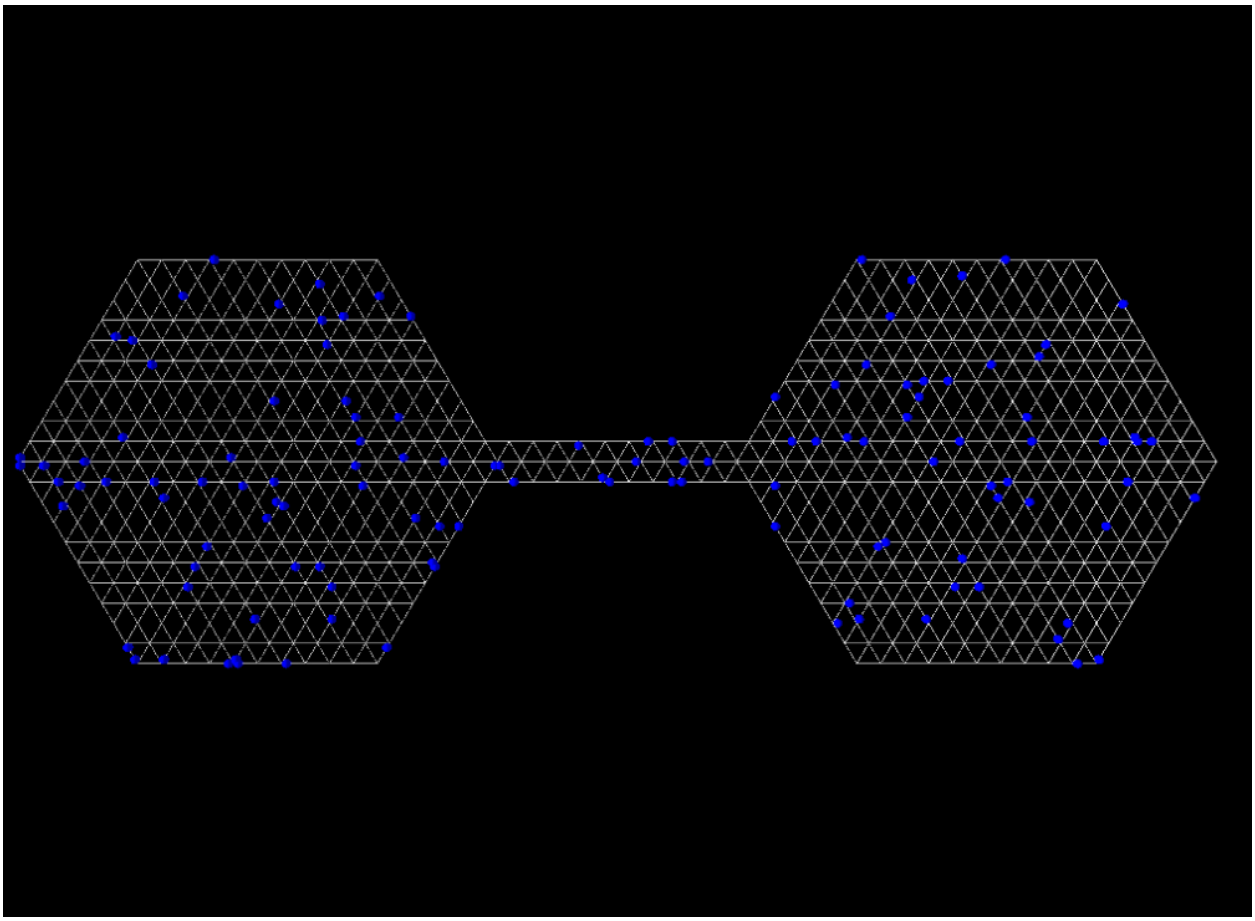


Figure 8: Experiment 1: The particles have diffused evenly throughout both containers.

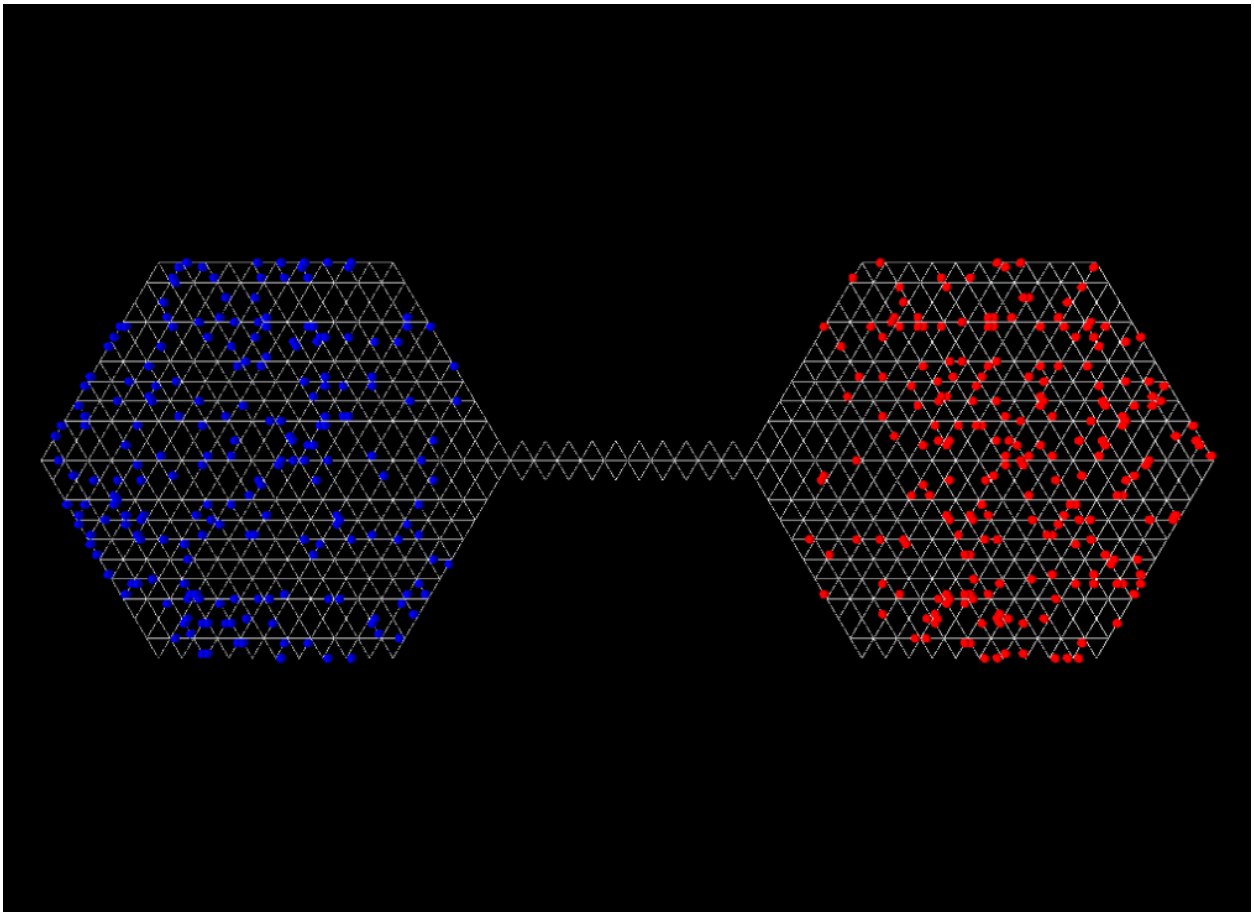


Figure 9: Experiment 2: Two distinguishable sets of particles begin from opposite containers.

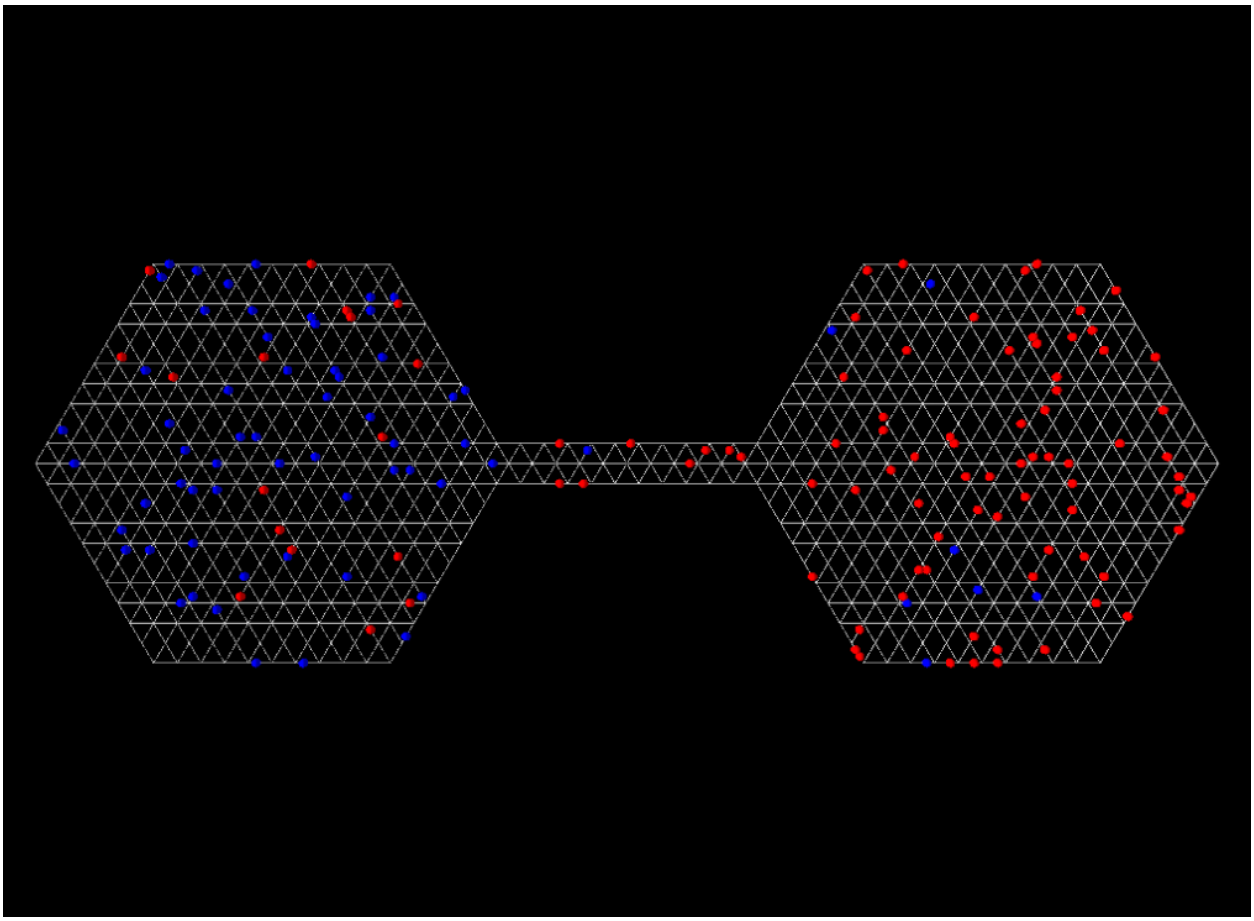


Figure 10: Experiment 2: It takes a considerable amount of time for the two sets to mix to any extent. (Compared to the diffusion time of experiment 1)

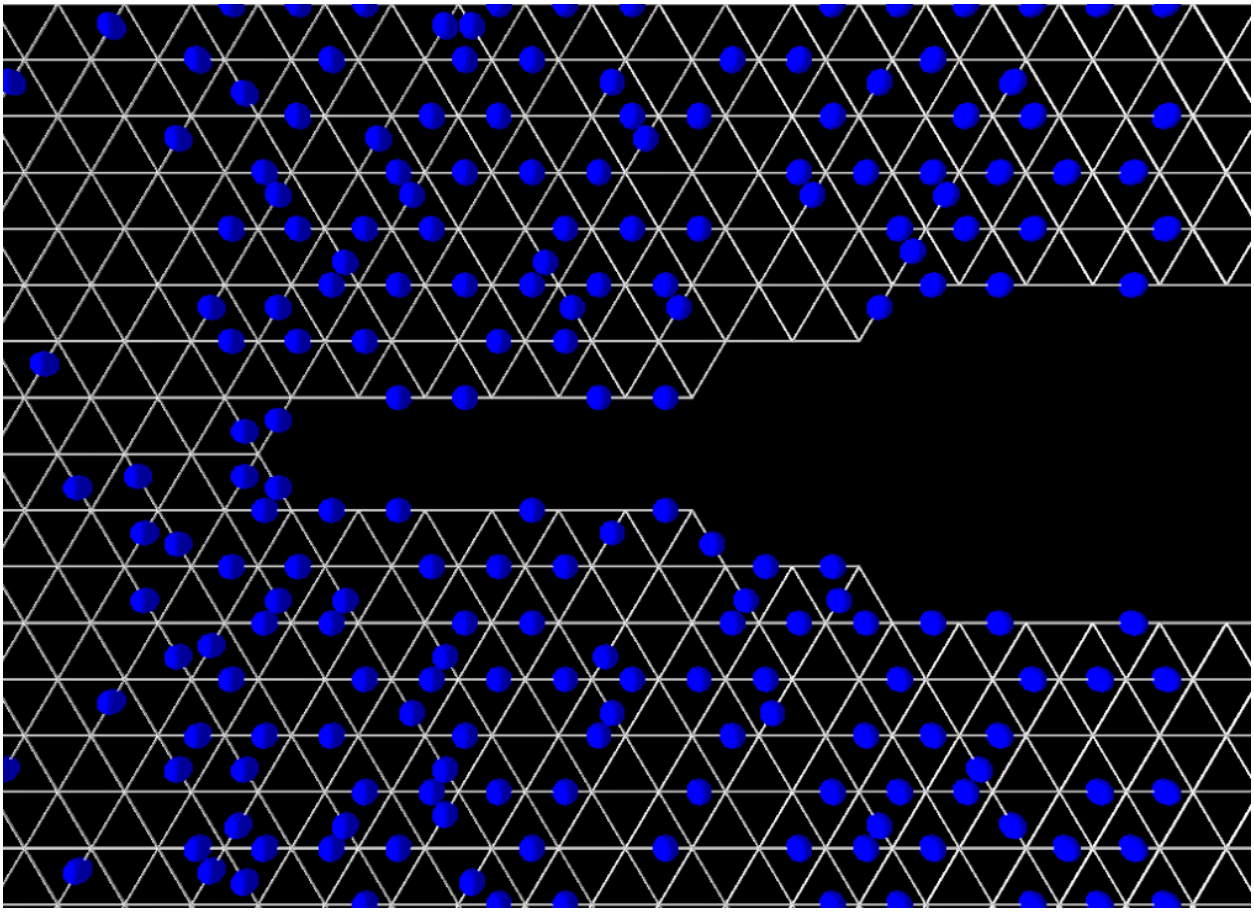


Figure 11: Experiment 3: Attempted to see if interesting boundaries form along the edges of a pin-like obstacle in the way of laminar flow.

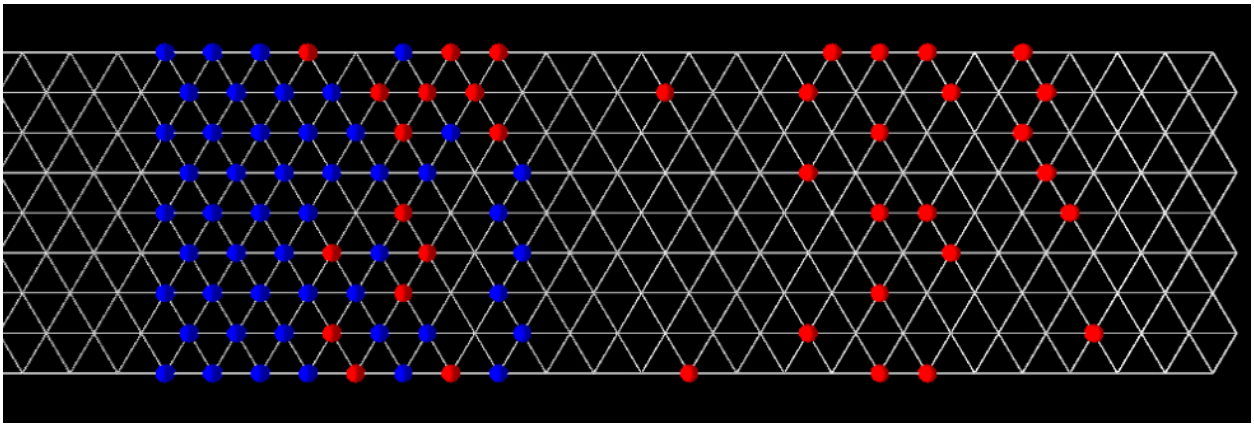


Figure 12: Experiment 4: A group of blue particles with net momenta to the right collides with a collection of red particles whose velocities are distributed randomly.

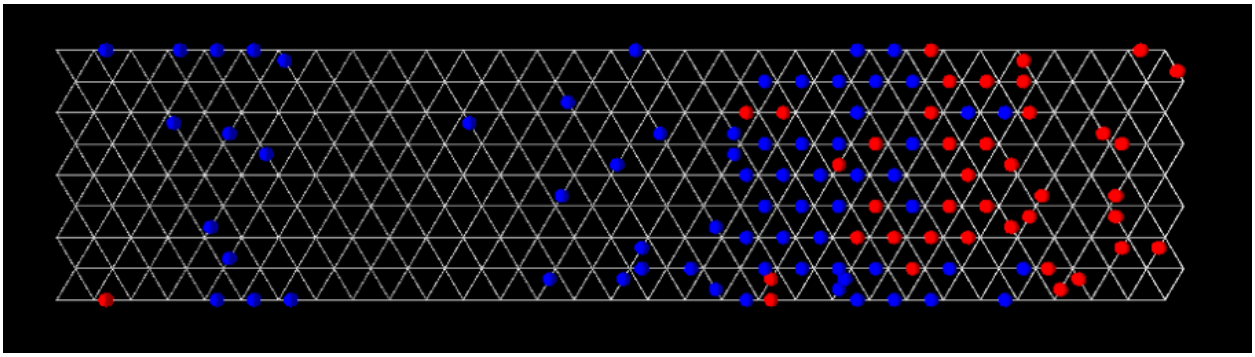


Figure 13: Experiment 4: The aftermath of such a collision; the red particles are "pushed" to the right.

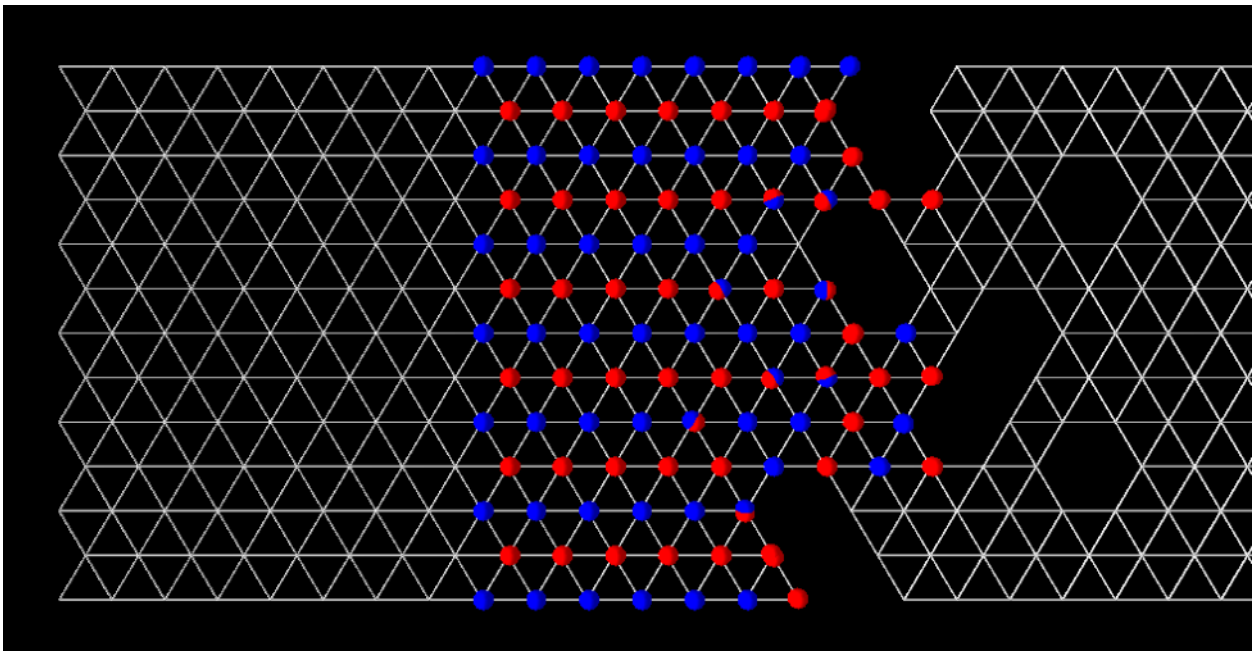


Figure 14: Experiment 5: A well-ordered, alternating-color rows of particles all move to the right towards obstacles.

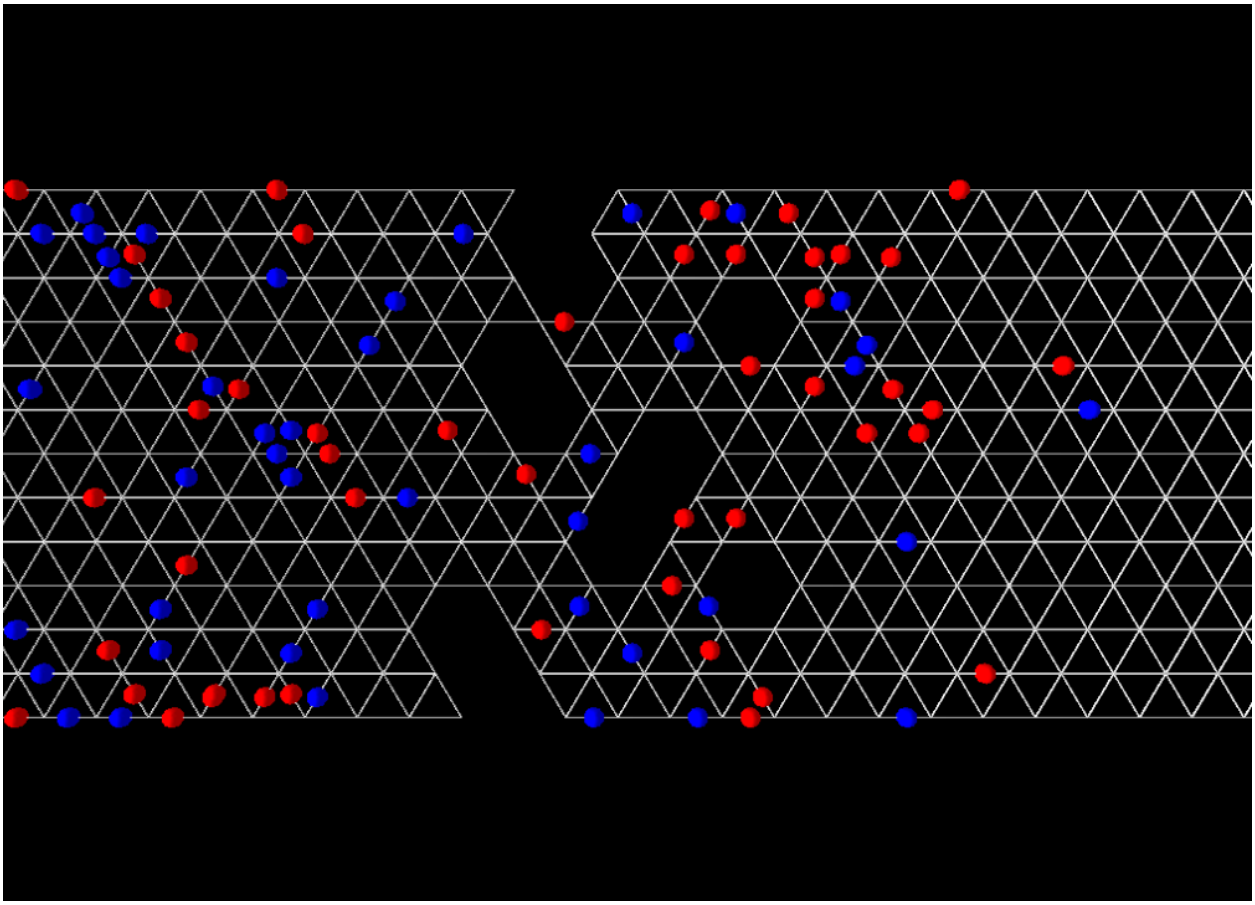


Figure 15: Experiment 5: The resulting mess after the main body makes past the obstacle.

References

[1] In developing the NodeManager algorithm used in the application, I benefited greatly from discussions with high school friend and current CS/math student at the University of British Columbia, Karl Dray. (karldray@interchange.ubc.ca)

[2] Josuttis, Nicolai M., *The C++ Standard Library*. Addison-Wesley, Indianapolis, 1999

[3] Luna, Frank, *Introduction to 3D Game Programming with DirectX 9.0c*. Wordware Publishing, Inc. Plano, Texas, 2006

[4] Rothman and Zaleski, *Lattice-Gas Cellular Automata*. CUP, Cambridge, 1997.

[5] Prata, Stephen, *C Primer Plus*. Sams Publishing, Indianapolis, 2005