Computational Fluid Dynamic Solver based on Cellular Discrete-Event Simulation for use in Biological Systems

By

Michael Van Schyndel, B.Eng.

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Science in Biomedical Engineering

Ottawa-Carleton Institute for Biomedical Engineering (OCIBME)

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

January 2014

Abstract

The use of computer simulations in biology and the medical research field has gained in popularity. These simulations are providing researchers the opportunity to better predict the behavior of biological systems before performing long and expensive physical trials. The modeling of large biological systems would benefit from a method of approximating fluid flow quickly and accurately. Currently, no analytical solution exists; instead, many different numerical methods attempt to provide accurate approximations. They are referred to as Computational Fluid Dynamic solvers (CFD).

The Discrete Event System Specification (DEVS) has rarely been used for modeling the physics of fluid flow. In this thesis we show how Cell-DEVS, a derivative of the DEVS formalism that conforms to the Cellular Automata parameters, can be used to provide realistic approximations of fluid flow. The algorithms used in the CFD presented in this thesis are based on the Navier-Stokes equations for non-linear fluid flow, which are an extension to Newton's Second Law of motion. The goal of the Cell-DEVS based CFD model will be to accurately approximate the fluid flow with minimal computational effort. Furthermore, the design of the solver should be such that it can be easily adjusted for use in a wide range of biological systems.

Acknowledgments

I would like to thank my supervisor, Dr. Gabriel Wainer. His dedication, support and advice has helped me tremendously over the last two years of my graduate studies and made them a rich and rewarding experience. I would like to thank Rhys Goldstein, Azam Khan and Jos Stam at Autodesk for his help and support during the last year. Finally, I would like to thank my family and friends, whose love and support has helped guide me through my entire graduate studies.

Contents

List of Tables

List of Figures

List of Equations

1 Introduction

1.1 Overview

The use of simulations has become a popular and powerful tool among medical researchers. The results generated by simulations can be used to complement experimental data and can be used to further develop a hypothesis. The use of simulations can aid the researcher by providing theoretical results that can be used to improve upon an existing hypothesis. These simulation results can then be validated by *in vitro* and *in vivo* studies.

Currently, many different types of simulations are used in a wide range of medical fields. For example, simulations are used to model forces generated by artificial hip joints. The focus of this thesis is the modeling of fluid flow within biological systems. These flows can be anything from blood circulating through the body, to air flow in the respiratory system or the flow of fluids in cartilage joints. Such a model is referred to as a Computational Fluid Dynamic solver (CFD). CFD solving is the process of calculating and describing the physics of the movement and interaction of fluid flow with the use of numerical methods [1].

With the advancement of computer technology we now have the ability to run larger more complex simulations of biological systems. One approach to resolving complex systems is to represent them as a sum of more simple systems. The Discrete Event Specification System (DEVS) is a formalism that implements this approach. The DEVS formalism, as described in *Theory of Modeling and Simulation* [2], allows for multiple DEVS models to be easily coupled representing systems that are more complex, even if the component models exhibit different time advancement patterns. A key principle of DEVS is that a model, the computer code which pertains to a specific real-world system, is separated from the simulator, the computer program which advances time.

Although there are numerous applications of DEVS for artificial systems, environmental systems, and both biological and physical processes, the formalism has been rarely applied to the physics of motion in two or three spatial dimensions. Simulations involving moving gases, liquids, rigid bodies, deformable solids, or a mixture of substances are usually implemented using traditional simulation techniques. The primary rationale for DEVS-based models of 2D and 3D solids, as well as fluids in motion, is the ease with which they can be coupled with other DEVS-based models. For example, researchers and engineers in the field of biomechanics could simulate an implanted medical device by coupling a DEVS model of the device with a DEVS model of the surrounding tissue using the same simulator. Another benefit of DEVS is the formalism's support of different time advancement patterns [3]. For example, the medical device model might be based on an event-driven approach, whereas the tissue model might use either fixed time steps or time steps that shorten in response to fast motion.

Most CFD solutions make use of different cellular methods. One particular method, Cellular Automata (CA) theory [4] is a branch of discrete dynamic systems. In these systems the real-world space is represented by a cellular grid, where each cell is treated as a finite state machine. In the CA formalism time is treated in a discrete manner, triggering state changes in the cells. The new state value for the cell is calculated based on the value of their neighboring cells. The Cell-DEVS (Cellular Discrete Event System Specification) formalism [4] is a related formalism in which each cell evolves asynchronously using explicit timing delays, which helps improve the efficiency by removing the burden of unnecessary processing. This allows for a more efficient asynchronous execution, using a continuous time-base, without losing accuracy. In this methodology, each cell is represented as a DEVS atomic model that changes state in response to the occurrence of events in an event-driven fashion.

Cell-DEVS was originally introduced for the modeling and simulation of spatial systems; however, not until current research has anyone proposed using the Cell-DEVS methodology to implement physics-based CFD equations to simulate fluid dynamics. The continuous time-advance of Cell-DEVS can contribute to the seamless simulation of CFD, in comparison with the discrete timing in CA that lacks the smoothness of fluid flow. Finally, the formal I/O port definitions in the formalism allows for information to be exchanged between different atomic and coupled models.

A model that was capable of realistically describing the physics of fluid flow and implemented using Cell-DEVS would have a wide range of applications. As previously mentioned the DEVS formalism allows for models to be easily coupled, while the rule-based nature of Cell-DEVS makes it easy for models to be adjusted. This means such a model could easily be adapted for use in any system that has fluid flow which needs to be resolved. In the past the description of the flow of fluids was crude, unless the fluid flow was the main concern of the model, then a CFD algorithm would be created specifically for that model. Now, models can be created using realistic fluid flows with minimal effort. Between the CFD model and the DEVS formalism, there now is a framework for simulating large, complex biological systems.

1.2 Contributions

In this thesis, we will introduce a Computational Fluid Dynamic solver that will help fill the gap in providing realistic approximations for fluid flow in biological systems. The CFD solver will be implemented as Cell-DEVS model using the CD++ Toolkit. Aside from the fact that a DEVS based CFD solver is a novel application, it will also contribute to the modeling of biological systems as a whole. As discussed later, many biological models will benefit from having a method of providing realistic approximations of the fluid flow to improve their results.

Furthermore, the model presented in this thesis will be designed in such a way that it will be easy to adapt for use in a wide range of applications. This will be done by simulating the effect of narrowing coronary arteries during Coronary Artery Disease, and observing the effect on the flow of blood to the heart muscles.

The following list of publications describe the algorithms presented in this thesis, the evolution of the algorithms to work as a Cell-DEVS based model and the implementation of the model with the CD++ toolkit:

- Michael Van Schyndel, Mohammad Moallemi, Gabriel Wainer. *Computational Fluid Dynamic Solver Based on Cellular Discrete-Event Simulation* In *Proceedings of SIMULTECH 2013*, Reykjavik, Iceland, 2013.
- Michael Van Schyndel, Gabriel Wainer, Rhys Goldstein, Jeremy Mogk, Azam Khan. *On the Definition of a Computational Fluid Dynamic Solver using Cellular Discrete-Event Simulation*. To appear in a special edition: *Journal of Computational Science: SIMULTECH 2013.*
- Michael Van Schyndel, Gabriel Wainer *Computational Fluid Dynamic Solver Based on Cellular Discrete-Event Simulation* to appear in *Proceedings of the Symposium On Theory of Modeling and Simulation-14*, Tampa, FL, USA

2 Review of the State of Art

Fluid dynamics is the study of the fluid mechanics involved in fluid flow. At any given point in our lives, we are surrounded by countless flows: the breeze blowing across your face, blood coursing through your body, or even the water running through the pipes beneath your feet. For a long time, we had no way to model these phenomena. Equations existed that tried to approximate the flows, however, the technology did not exist to make them of any real use until the advent of the computer. The computer was better suited to handle the intense amount of calculations required to implement these equations, and soon methods of solving these fluid flows were created. These methods were called Computational Fluid Dynamics (CFD). Today CFDs are used in labs and research to aid understanding of why fluids behave the way they do. They are used by engineers to help in the design process. In the video game industry, CFDs are used to create incredibly realistic physics for massive virtual environments. As the demand grows for CFDs so does the need for them to provide more realistic physics and faster. The possible applications of a CFD are only limited by ones imagination.

2.1 Simulating a Computational Fluid Dynamic Solver as a Cellular Automata

In simulations our goal is to create a model that can imitate or describe a real-world system. A good simulation is one whose results accurately match those of a real-world system under a certain set of parameters. If the model works well it can even be used to predict behaviors of systems and draw conclusions from these results without having to be performed in the real-world. With the advancement of computer technology came an increase in the use of simulations and the ability to create more complex models for more complex systems, with the key component still being accuracy. Fluid dynamics had always been a difficult system to model,

requiring a large amount of calculations that were impossible to perform before computers, even though there were equations predicting their behavior. Now with the aid of computers we are finally able to model fluid flows. These models are called Computational Fluid Dynamic (CFD) solvers.

There exist many approaches to solving for fluid flow. They may vary in the methods used in the calculations and their accuracy of their physics; however, they all have the same goal in mind. That goal is to create the most accurate model possible of real-life situations: whether it is examining the stresses on an aircraft wing at 50,000 feet or smoke rising from the tip of a cigarette in a video game. The method chosen varies and is based on many different criteria. For example, on the level of physical accuracy desired and the level of computational power available. Most models use some derivative of the compact Navier-Stokes equations, equations 1&2, to solve for the evolving fluids [5].

$$\frac{\Delta u}{\Delta t} = -(u.\nabla)u + v\nabla^2 u + f$$

Equation 1 Compact Vector Notation of Navier-Stoker Equation for solving the velocity field

The Navier-Stokes equations, named after Claude-Louis Navier and George Gabriel Stokes, make use of Newton's Second law by applying it to fluid flow, and assuming that the stress on the fluid is proportional to the diffusing viscous term and the pressure term [5]. The first equation is for solving the velocity vectors; the sum of which is hereafter referred to as the velocity field. The equation is a re-arrangement of the incompressible flow of Newtonian fluids. The acceleration ($\frac{\Delta u}{\Delta t}$) is equal to the sum of: the negative continuity equation ($(u.\nabla)u$, responsible for the conservation of mass), the viscosity ($v\nabla^2 u$) and any body forces present (*f*). In other words, the change in the evolution of the velocity field is based on the viscosity and any

other forces that may act upon it, such as a heating vent. While this is the most important part of any good CFD solver, it provides very little visually. To make it more useful, we must demonstrate particles moving through the velocity fields. To move objects, we must simply determine what forces are going to be acting on it and in what direction. These forces are extracted from the velocity fields. Most of the objects we wish to move are relatively light and the only relevant forces are those applied by the velocity field, such as dust or smoke [5]. One could simply apply these forces to the particles and see how they move, however, for more complex models it would be taxing to perform these calculations for a large number of particles. Instead, we could treat the matter as a density of particles, where instead of either being 0 or 1(no particle, particle respectively); we would treat it as a gradient value that ranges from no particles present to some maximum number of particles present. The forces on these densities are applied using equation 2, which is similar to the equation used for evolving velocities but more simplified since the only forces present are solely generated from the velocity vector field.

$$\frac{\Delta p}{\Delta t} = -(u.\nabla)p + k\nabla^2 p + s$$

Equation 2 Compact Vector Notation of Navier-Stoker Equation for solving the Density field

Cellular Automata (CA) theory existed well before computers were popular and its guiding principle being emergence [6]. Emergence is the idea of highly complex features being derived from rules and properties of low complexity. To understand what CA is start with an infinite grid space of cells, with (n) dimensions, which can be used to represent an enclosed universe or system. These cells can take any desired shape, for example squares, triangles or hexagons can be used for 2D spaces. Often with models we desire cell space to behave as a closed continuous system; i.e. if one were to travel through the right-side boundary they would appear at the left-side boundary. True to the principle of CA the solution to this is simple. Take

the grid space and wrap it so that the cells of the first column and the last column re neighboring and again so that the first row is neighboring the last row. The result is a torus; see figure 1, a seemingly continuous 2 dimensional grid space.

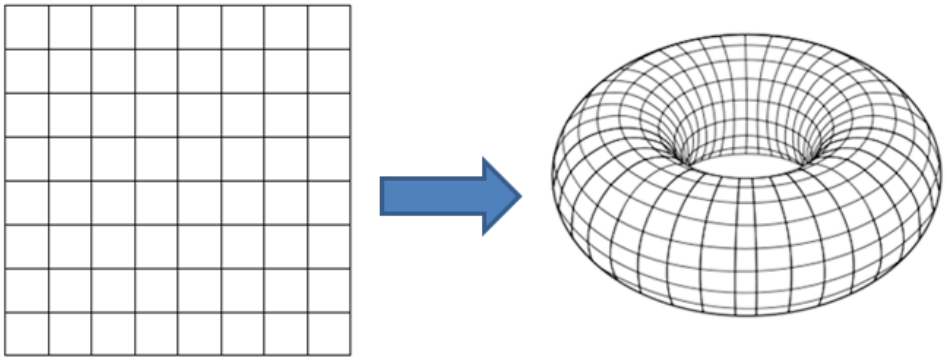

Figure 1: Wrapping a grid space to create a seemingly infinite system

The next characteristic is that after each time step, each cell must be a discrete state from a set of determined states. The cell's state is determined solely on the local neighborhood cells, and each cell contains its own local computing method, that is to say, the method for determining its new state.

The most basic of fluid model for a CFD implemented as a CA is the Lattice Gas Automata (LGA) [7]. The particles can travel in one of four directions through the square lattice, as shown in figure 1. Only one particle is allowed to enter any given site for any given direction. When modeling such a system, we usually evolve it in two steps: the collision and the movement. For this model 4-bits of information is all that is required to represent the changes during the movement phase. Each bit is used to represent a particle moving in from a pre-defined direction. The first bit could represent a particle entering from the bottom, second bit from the left and so forth.

This simple method of modeling is a prime example of the Emergence theory of CA. From a series of case definitions to determine the new state of a cell, we are able to create a complex model. As seen in figure 2, this method is suitable for modeling the diffusion of simple particles through a given space. By assigning random initial movements and defining the behaviors of particle collisions, the particles eventually diffuse to take the shape of the container. While the results behave similar to the diffusion of gas molecules in a closed space, it is still far from a functioning fluid dynamic solver capable of simulating real fluid flow. That is to say, no matter how complex we define the movements, 4, 6 or 9 directional, and no matter how complex we make the definitions for particle collision, it will still require a method for adding external forces to the particles.

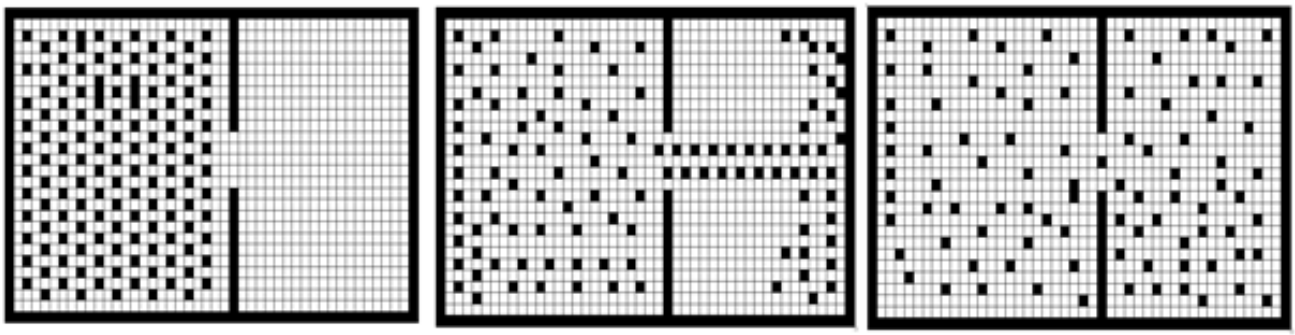

Figure 2: Particles dispersing in a 6-direction LGA over increasing periods

The model could be further adapted for a hexagonal or triangular grid space and have six possible movement directions instead [8], as seen in figure 3. The added directions of motion allow movements that are more realistic. The number of bits required increases to six, once again with each bit representing a specific direction of travel. With the increased number of directions comes an increase in the number of cases that must be defined to determine the new cell states. Instead of having to only worry about two and three particles collisions, now cases must be made

for four and five particle collisions as well. Add the fact that there is more than one way for two or three particles to collide with each other, this gives rise to a significant number of cases will be required to handle all the particle-particle and particle-barrier collisions.

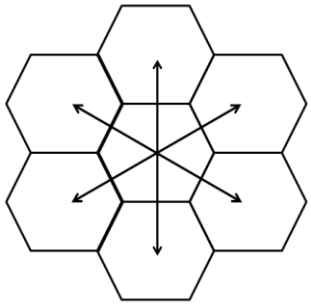

Figure 3: A hexagonal grid with motion in six directions

Finally the most advanced of these models is the Lattice-Boltzmann method (LBM) [9]. Similar to before, it is comprised of two steps: collision and movement. The more advanced models return to a square grid space with eight directions of movement and a rest state, see figure 4.

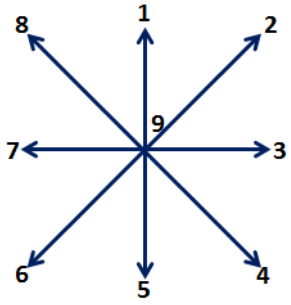

Figure 4: Orthogonal directions for Lattice-Boltzmann method

With additional movement directions comes an increase in the number of possible collisions whose behavior must be defined. Instead of the random movement used in previous

methods, the movement is now calculated from a velocity field, which is determined using one of several computational methods, which one depends on the exact nature of the model. For example, an LBM for turbulent flow would either make use of the *direct numerical simulation* or *large-eddy simulation* [9]. The *direct numerical simulation* (DNS) method is when the Navier-Stokes equations are numerically solved without any turbulence [10]. The range of scales required to accurately compute the flow is based on the physics of the fluid and represented as the *Kolmogorov length scale*. Moin et al. showed that it was possible to produce results with larger scales than the *Kolmogorov* numbers. The DNS method is computationally intensive and is best suited for aero acoustics, high-speed flows and reacting flows [10]. Large-eddy simulation (LES) work by reducing the length of the scales required to compute the Navier-Stokes equations. This is done by applying a low-pass filter to the equations to eliminate the smaller scales of the solution. This method resolves the larger scales of the velocity field and decreases the computational cost. The range of the small scales that are filtered can be adjusted to reflect the desired resolution and the computational power available [11]. The LES method has been successfully implemented to simulate turbulent combustion [12] and in the simulation of the stable atmospheric boundary layer [13].

As previously mentioned, there are wide varieties of CFD solvers that make use of an equally wide variety of methods. One particular algorithm was created for theoretical use in video games and it was based on the Navier-Stokes equations [5]. The algorithm was an attempt to create a realistic CFD with minimal computational effort since it would be primarily run with an average PC or gaming console; it should be fast, realistic and not consume too much power. The algorithm was not created as a Cellular Automata; however, it did share some similar characteristics. The system was broken into a grid space of square cells, where the density and

velocity information was stored at the center of the cells. The new cell states were calculated based on the surrounding cells, however the neighborhood was not rigid. While the cells were updated synchronously, there were a large number of calculations to be performed between time steps before the cells were updated. Additionally multiple layers were required to store additional information, instead of being stored in a single cell.

While Cellular Automata models are simulated in discrete time steps and have discrete state values, others may benefit from the use of changing time steps. In many biological systems a single event may trigger a cascade of events [14]. For example, when simulating a neuron the state remains constant until an input/event occurs. This will cause a series of state changes to occur until once again the neuron returns to its resting state. At this point it could be a significant period of time until the next event, in which case it would be exhaustive to be constantly trying to update the cell. Similarly, the speed at which an event occurs may differ significantly. When breathing the flow of air is greatest during the inhale and exhale, while dropping very low in between the events. With a lower flow rate it would not be necessary to have the same time resolution during the resting period as that of the inhalation or exhalation period. This method of simulation is referred to as “discrete event simulation”.

2.2 DEVS

The Discrete Event System Specification (DEVS) came about as a way to model such types of simulations. The formalism was first introduced in 1976 by Bernard P Zeigler [2] as an extension of the Moore machine formalism. This was done by adding a lifetime associated with each cell state as well as adding a hierarchical method called *coupling*.

The building block to any DEVS model is the Atomic Model. The atomic model is defined as a septuple, as follows [15]:

$$M = < X, Y, S, ta, \delta_{ext}, \delta_{int}, \lambda >$$

Equation 3: DEVS Atomic model definition

Where:
X is the set of input events

Y is the set of output events

S is the set of sequential states

ta is the time advance function

δ_{ext} is the external transition function

δ_{int} is the internal transition

λ is the output function

2.3 Cell-DEVS

Cell-DEVS is an implementation of the DEVS formalism that conforms to the parameters of a Cellular Automata [16]. The definition for a Cell-DEVS atomic model is as follows [15]:

$$TDC = < X, Y, I, S, N, delay, d, \delta_{ext}, \delta_{int}, \tau, \lambda, D >$$

Equation 4: Cell-DEVS Atomic model definition

Where:

X is the set of external input events

Y is the set of external output events

S is the set of sequential states

I is the model interface

N is the set of input events

delay defines the type of delay for the cell

d is the duration of the delay

δ_{ext} is the external transition function

δ_{int} is the internal transition

λ is the output function

D is the state duration function

As before, these atomic models can be combined in a hierarchical method as a coupled model. The definition for a Cell-DEVS coupled model is as follows [15]:

$$GCC =< Xlist, Ylist, I, X, Y, n, \{t1, \ldots, tn\}, N, C, B, Z >$$

Equation 5: Definition for a Coupled Cell-DEVS model

Where:

Xlist is the input coupling list

Ylist is the output coupling list

I is the model interface

X is the set of external input events

Y is the set of external output events

n is the dimensions of a cell space

{t1... tn} is the number of cells for each dimension

N is the neighborhood set

C is the cell space

B is the boarder cells

Z is the translation function

The atomic models are arranged in an array with each being connected to its neighbor and the **Z** function defining the internal and external couplings.

2.4 Implementation of Cell-DEVS models using CD++

Cell-DEVS models are built using the CD++ toolkit. The toolkit includes a high-level scripting language, a simulation engine, a testing interface, as well as a basic 2D and 3D visualization of the simulation results. Figure 5 shows an atomic Cell-DEVS model in its most basic form:

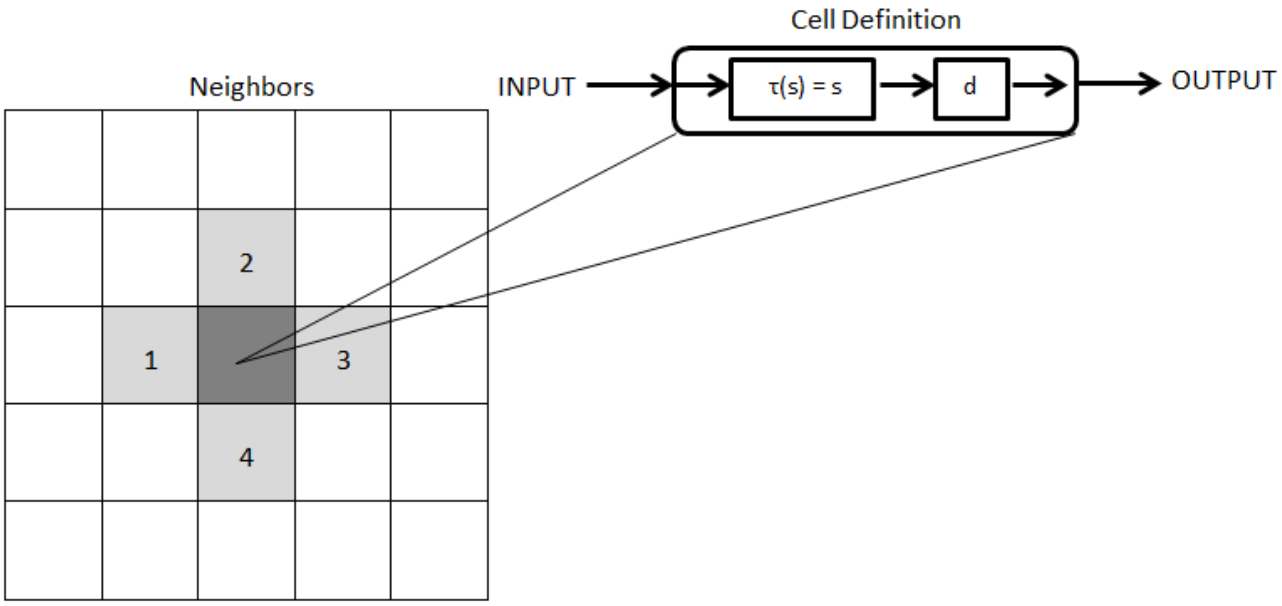

Figure 5: Cell-DEVS Model

Each cell of the lattice contains information regarding its neighborhood and its local computing function. This local computing function has 3 main parts; **PostCondition, Delay** and **PreCondition**. That is to say, when defining the local computing function, simply define the **PreCondition** that must be satisfied so that the **PostCondition** will be applied to that cell after the **Delay** has expired. By using a source-destination method of evaluating the cells, Cell-DEVS allows for the cells to be calculated asynchronously and then updated all at once. This feature allows for the possibility of parallel computing.

The CD++ Toolkit provides a high level scripting language for writing Cell-DEVS models. By using the toolkit, the user can define models and save tedious coding. A Cell-DEVS coupled model is defined by equation 5. During the initialization of the model several of the terms are defined, such as; **n**, **{t1... tn}**, **N**, **C** and **B**. The following code is a sample of the implementation of the initialization code:

```
[top]
components : cfd

[cfd]
type : cell
width : 75
height : 25
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : cfd(-1,-1) cfd(-1,0) cfd(-1,1)
neighbors : cfd(0,-1)  cfd(0,0)  cfd(0,1)
neighbors : cfd(1,-1)  cfd(1,0)  cfd(1,1)
initialvalue : 0
localtransition : conrad-rule
neighborports : value diffusion u v boundary p div
```

The first thing to appear in the code is the components used in this model. If the model were a system of atomic models, a coupled model, this section would be the first step to coupling the atomic models and defining the hierarchy. For this sample, the atomic model is named *cfd*. After defining the atomic model we set the cell dimensions, **n**, and the number of cells per dimension, **{t1... tn}**. This model will only have two dimensions with the width being 75 and height as 25. The next important step is to determine the behavior of the boundaries, **B**. As previously discussed, it is often important that the system be continuous along all the boundaries, as described in section 2.1. If such is the case, the user selects the *wrapped* option for the boarder, as done here. The other option is to have *unwrapped* borders, in which case the space beyond the borders is treated as undefined. If this is the case, special care must be taken to ensure the model does not attempt to access information beyond the borders, as this could lead to errors

in the simulation results. The principle of Cell-DEVS is based on the idea that during the evolution of a model, the values used in the local computing function are taken from the defined set of surrounding cells called the neighborhood. This makes the definition of the neighborhood, **N**, one of the more important steps when initializing a cell [5].

With the new Cell-DEVS simulator came two new functions that were beneficial to developing models: the addition of *variables* and *neighborport*, hereafter referred to as ports. Variables are useful for storing information locally, to be accessed by the cells during the local computing function calculations; however, the information stored within them could only be accessed by the cell to which it was linked and not by any neighboring cells. This causes complications when it becomes necessary to share the information between cells that is stored within a variable. The *variables* method is useful when additional information storage is required that is only accessed locally and can drastically improve the computational load. Alternatively, with neighbor ports, the information is stored in variables, similar to before; however, these variables are accessible to any of the neighboring cells via ports. Both methods improved the computational efficiency significantly for the model. The following is an example of how ports and variables are initialized:

```
neighborports : value count1 count2
statevariables: value count1 count2
```

By this point, all that remains is defining the local computing function. The local computing function is the set of rules located in each cell that governs cell behavior and determines the state changes. The local computing function is defined as consisting of a **PostCondition, Delay** and **PreCondition** statement. The following is an example of such a rule:

```
rule : { ~value := if((0,-1)~value = 1, 0,1); } 100 { (0,0)~value = 1 }
```

Here the **PreCondition** is that the state of the *value* port must be equal to one. If this is true then the state of the *value* port is equal to zero if the neighboring state is one else it would retain the state value of one. This is the **PostCondition**. Finally, the **delay** is set; in this case a delay of 100ms was chosen.

The models that have been implemented using the CD++ Toolkit have varied widely in their complexity and application. In most cases the results of the simulations were able to provide useful insight to their related fields. For instance, a model for evacuation of a building presented in [17], as seen in figure 6, is a clear example of how, by following the CA parameters and implementing simple local transition functions, a complex model can be created. The model included human behavior for the evacuation of a building with multiple floors.

The building evacuation model could be broken into three distinct steps. The first step is the initialization step. During this step the walls and obstacles are added, as well as the emergency exits and the people. The next step is the mapping stage. Using an iterative process a map is overlaid on the floor plan that determines the minimum number of steps it would take to reach an exit from that cell. This map defines the routes that people will take to reach an exit by ensuring the people are always heading to a cell with fewer steps to an exit. The final step was the evacuation itself, were the people moved through the building towards the front doors on the first floor.

The work done by Wang et al.[17] was to create a framework to integrate specialized software to improve the real-world applications of the simulation. The building plans were extracted from *Revit*, a software specializing in the designing of building. These building plans were loaded into the modeler, where I simulation was executed. These results were then parsed and loaded into a second specialized software, *3Ds Max,* were the original blueprints were

loaded and the results superimposed over the building, see figure 6. Overall this allowed for more realistic simulations to be created with results that have an impact on real-world decision making. Finally, these results were visualized in a more meaningful manner. A more detailed explanation can be found in Appendix A.

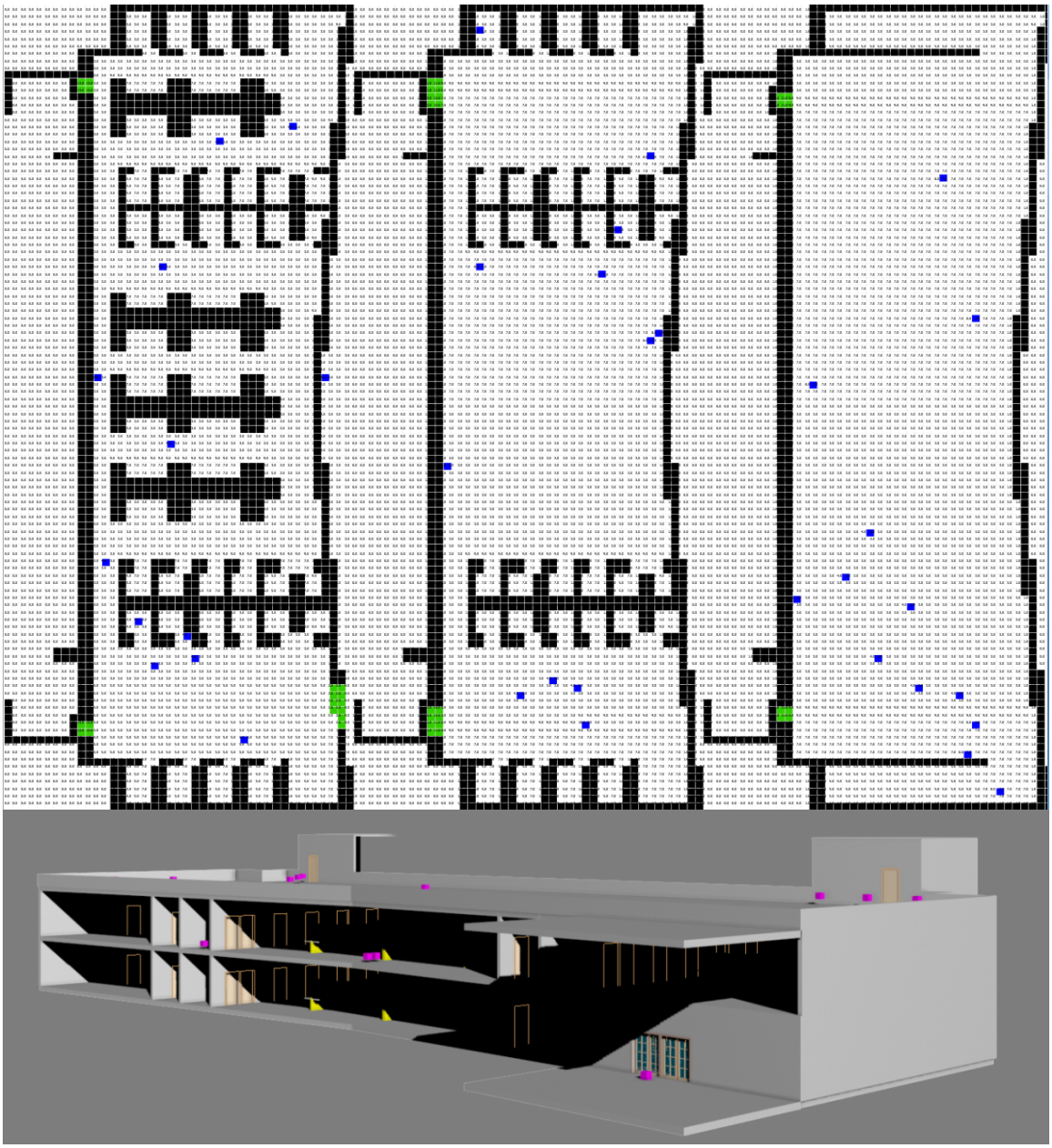

Figure 6: Building Evacuation Model of a 3-Story Building [17]

The model created was flexible and could be applied to any building blueprint. By using real building plans it was able to provide useful feedback by determining flaws in the building design such as: chokepoint, obstacles and lack of necessary emergency exits, which would be beneficial to the design process for a building. Other useful feedback included the ability to estimate the evacuation time based on the number of people present, which could in turn help to establish a maximum occupancy number to ensure sufficient time to evacuate the building.

2.5 CFD and Cell-DEVS modeling of Biological Systems and the Medical Field

Even with a rising increase in DEVS formalism and simulations being used in the field of biological systems there has been very little cross-over between the two. Wainer ET al. presented several models where both DEVS and Cell-DEVS were used to create biological models, including a model of a human liver and synapsin and vesicle interactions in a neuron [18]. One work looked at the effect immune cells played on the growth of a tumor mass, see figure 7 [19]. The model design is that of a Cell-DEVS with the transitional states representing different physiological conditions. First, the cells were either tumor-related cells or general immune cells responsible for attacking the tumor. The tumor cells fell into one of three categories; proliferative, dormant and necrotic. By adjusting the saturation of immune cells, the model demonstrated how the immune cells were able to slow and or stop the growth of a tumor as seen in figure 7.

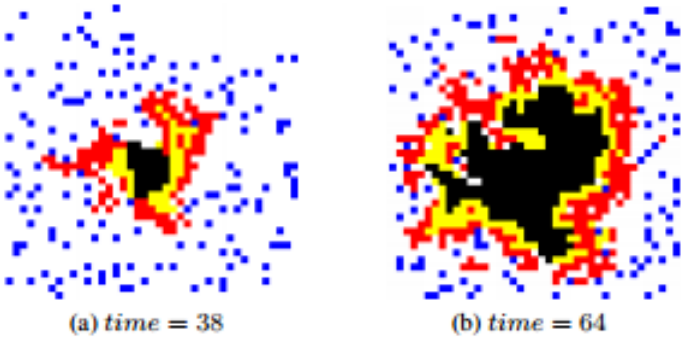

Figure 7: Snapshot of Tumor Model [19]

Figure 8 is a snapshot of a Nerve Synapsis model that demonstrates the binding of Synapsin, the small particles, to the vesicles, larger particles, to form vesicle clusters that bound to the active zone, bottom of the snapshot, of a presynaptic nerve terminal [20].

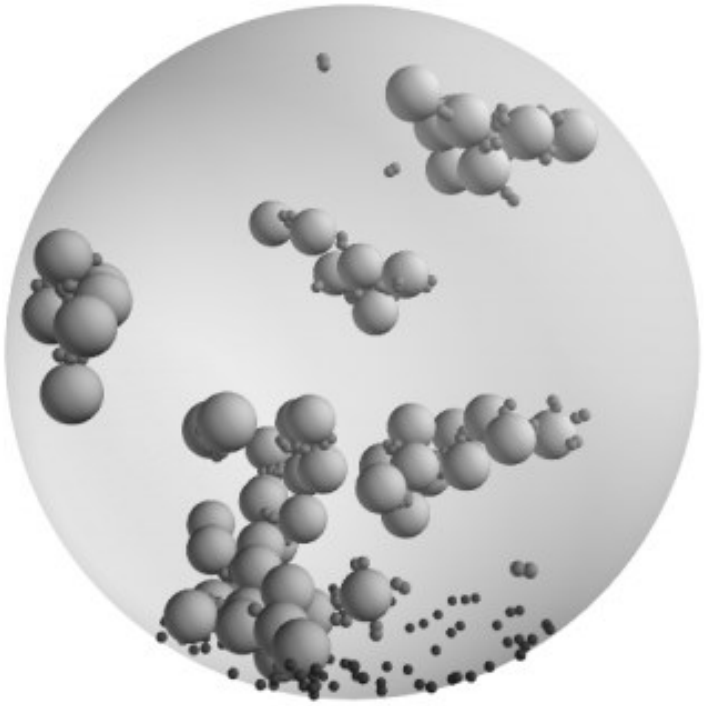

Figure 8: Snapshot of Nerve Synapsis Model (used with permission)[20]

Most biological systems are complex and broken into definable sub-units. The DEVS formalism would allow these sub-units to be represented as individual atomic models, and with the hierarchal nature of DEVS would allow these atomic models to be coupled in order to model the entire system [21]. These atomic models can be re-used in other systems, when applicable, with relative ease. This ability of DEVS to represent large systems as a sum of smaller definable parts makes it a viable option for future modeling of large biological systems.

Huang et al. proposed using a CFD approach to provide kinetic information for biological systems. They found that enzymatic reactions are controlled by the availability of the reactants. These reactants are transported via a biological system. To implement this idea they modeled a biological wastewater treatment system. This system helped provide information regarding the metabolism of organic carbon substrates and populations of microbial. Their findings were that, by coupling a CFD model they were able to produce more accurate reaction kinetics and provide more realistic kinetic models for biological systems [23].

Aside from being coupled with biological systems to improve the calculations of flows, CFDs have further use in the medical field. For instance, Scott et al. examined the role a CFD model might play in the Biological Safety Environment. The goal was to provide air flow analysis and air-born contaminant tracking [24].

In a similar field, Xu et al. looked at how a CFD model could be coupled to evaluate the design of hospital UV Germicidal Irradiation Systems (UVGI) for the elimination of airborne mycobacteria [25]. The hypothesis was that the system was most effective when no airflow was present, and therefore they wished to model different room parameters that would improve the effectiveness of the system. They found that a CFD model was useful in judging the effectiveness of a hospitals UVGI system and ventilation system in infection control [25].

Finally, the Applied Research lab at Pennsylvania State University has a project looking at biological and biomedical flows. One project combined 3D imaging technology with a CFD model to simulate the oxygen uptake in the human respiratory system [26]. Other research includes the modeling of the fluid mechanics of white blood cells and how they interact with cell walls and cancerous cells [26]. CFDs were also used to analyze several heart-assist devices, small pumps that are used to aid in the circulation of blood [26].

Currently there exist no DEVS or Cell-DEVS based Computational Fluid Dynamics solvers aside from the work being described in this thesis.

3. Problem Statement

The problem addressed in this thesis, is the need for a Computational Fluid Dynamic solver to be implemented using the Discrete Event Systems Specification for use in solving dynamic fluid flows in biological systems. This CFD solver needs to provide realistic approximations for the behavior of fluid flow with minimal computational effort. The solver needs to be versatile so it can be used in a wide variety of applications. With no analytical solutions existing for non-linear flow, numerical approximation methods are used to describe their behaviors. Historically, these methods tend to be complex and require a lot of time and effort to simulate, and therefore CFDs are mostly used when we wish to observe only the behavior of the fluid. In biological systems, such as the human circulatory system, the flow of the blood plays a large role in the overall behavior of the system; however, it may not be the focus of the simulation. Having a versatile model that would be able to approximate the behavior of the blood flow with minimal effort would allow more complex biological models, such as the human circulatory system, to be created.

The application of CFDs to biological systems is relatively new, with very little work being done. This could be partly because the CFD solver would have to be re-written for each application. The Cell-DEVS formalism provides a method where models can be easily re-used for a wide variety of applications with little to no changes required to be made to the model. In addition, the hierarchal method of coupling models allows for the combining of many basic atomic models, each representing a specific part of the system, to be combined and simulated to solve for large systems that could otherwise not be modeled.

At the beginning of the project two choices were made. The first was that the CFD model would behave as a Cellular Automaton and be implemented as a Cell-DEVS atomic model. The

second was to choose to model the fluid flow using the Navier-Stokes equations and specifically to make implement the theoretical method for solving the Navier-Stokes equations outline in [5]. To summarize the problem being solved: a Computational Fluid Dynamic solver implementing the Navier-Stokes equations to solve for the fluid flow, implemented as a Cellular Automata using the cell-based Discrete Event Simulation Specification formalism (Cell-DEVS) to provide a method for modeling dynamic fluid flow in biological systems.

Already there exist a large number of CFD solvers that make use of the Navier-Stokes equations to solve for fluid flow, each implemented in its own unique way. As previously stated, the goal of these CFD methods are to provide the most accurate and detailed results since it is most likely the behavior of the fluids that is being studied. Because of this, these methods are slow and cannot be executed with the average computer. The goal of the algorithms outlined by Stam [5] was to provide the highest level of detail with the lowest computational cost. This is important, since the goal is to create a model that can be integrated as a part of a greater whole; for example, an entire biological system. Since the specific behaviors of the fluids themselves are not as important as the overall behavior of the system the simulation of the fluids should be as fast as possible.

Prior to the work in this thesis, the DEVS formalism had never been applied to the issue to solve for the dynamics of fluid flow. DEVS based models such as those presented in [18], [19] and [20] may have benefited from an accurate method for calculating the forces generated by fluid flow. Previous methods to handle movement within a fluid space had been crude at best.

The "Computational Fluid Dynamic Solver based on Cellular Discrete-Event Simulation for use in biological systems" is a novel problem whose solution would fill a much needed gap. Although the DEVS models described in section 2 demonstrate the capabilities of the DEVS

formalism, they are relatively basic. They usually consist of no more than a few atomic models and the behavior of the movement through a fluid space is often a crude approximation of the physics of real-world behaviors of fluids simply because the fluid flow is not the focus of the simulation. For example, the tumor model presented in section 2.5 describes the movement of the white blood cells as random, making contact with the tumor by chance. A coupled model could be created were the CFD model provides the behavior of the circulatory system and provides the necessary forces to solve for the movement of the white blood cells. This would provide more significant simulation results.

The application of this model will give rise to possible development in a wide range of areas. These areas would include such fields as: 3D visualization of DEVS based models [17][20], the integration of the DEVS simulator with other software's to provide real-world applications for the simulations such as the work presented in [17]. In addition, integrating the DEVS based simulator with databases of three dimensional models, such as the Parametric Human Project [27], to model complex biological systems by coupling more basic DEVS atomic models. This project also provides further evidence that the fundamental principle of Cellular Automata, emergence, is true and allows for the possibility of even more complex models to be implemented as CA.

Even where the project may reveal deficiencies in the model, they are such that they open a new avenue of discussion. The knowledge gained in this thesis can be used as a foundation for developing a better implementation of a CFD. Future works can refer to this project for examples of how to re-create what is a complicated set of calculations as a simple set of rules. The framework for how to handle fluid flows in CA will be useful to many fields of study.

4. Model Definition & Implementation

In this chapter we will be describing the implementation of the model presented in this thesis as a Cell-DEVS model using CD++ Toolkit. We will begin by presenting the Navier-Stokes based algorithms that are used to approximate the behaviors of fluids. Then we will describe how these algorithms are implemented to create the rules that will define the behavior of the model.

4.1 Model Definition

The algorithms being presented in this thesis can be conceptually broken into two main parts: the density solver and the velocity solver. Each of these sections is responsible for resolving one of the two Navier-Stoker equations, equations 1 and 2. Both equations can be further broken into discrete steps (functions) that resolve individual terms within the two equations, as seen in figure 9 and 10.

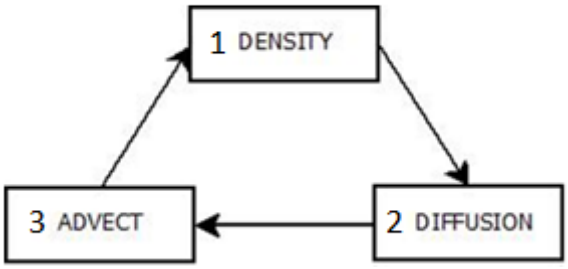

Figure 9: Density Solver Steps

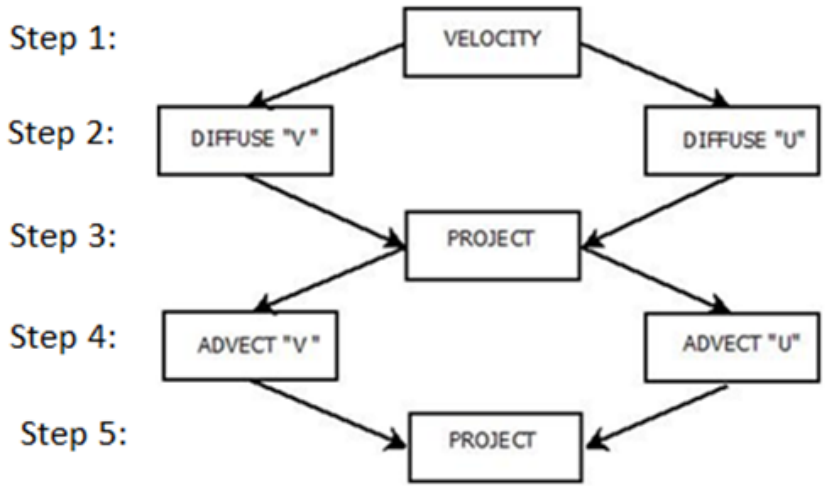

Figure 10: Velocity Solver Steps

As seen in figures 9 and 10, several of the functions appear in both solvers. This will be important when we implement the model since functions can be re-used in both solvers.

In figure 9, the density solver is the more basic routine with just the three steps: diffuse, advect, and update. The two functions used are the *diffusion* and the *advection* functions. The *diffusion* function is responsible for resolving the radiation of the system and is solved using the following equation:

$$\boldsymbol{x(i,j)} = \frac{[\boldsymbol{x(i,j)'} + \mathbf{a} \times [(\boldsymbol{x(i-1,j)} + \boldsymbol{x(i+1,j)} + \boldsymbol{x(i,j+1)} + \boldsymbol{x(i,j-1)})]]}{\mathbf{1} + \mathbf{4a}}$$

Equation 6: Diffusion Calculation

Equation 6 states that the new density in the cell at position (i, j) is equal to what will remain in the cell from the original density, x'(i,j), plus what will enter the cell from the four cardinal neighbors, x(i-1,j), x(i+1,j), x(i,j+1), x(i,j-1). The rate at which the density moves between the cells is thought of as the viscosity and is represented by the variable a. To increase the resolution of the model this step is run multiple times [5].

Next, the *advection* function is responsible for applying the forces generated by the velocity fields. The information for the forces is stored in two ports: the horizontal velocity component (u) and the vertical velocity component (v). Therefore, the force acting on the density at any location is equal to the magnitude of the respective velocity component vectors, *u* and *v*.

To apply the forces is significantly more complicated. The first approach would be to say, if this is the force acting on me I will end up here. However, since the system is treated as a cell space and the densities are thought to exist in the cell center, when advected it is highly unlikely that the destination lies at the center of the new cell. To resolve this issue it would be possible to then average the density out to the 4 surrounding cells, but this is method is complicated and prone to instability; instead, a more simple approach exists. To move the density, simply trace backwards from the cell center to determine where the density would have to come from to end up exactly in the center of the current cell space, as seen in figure 11 [5].

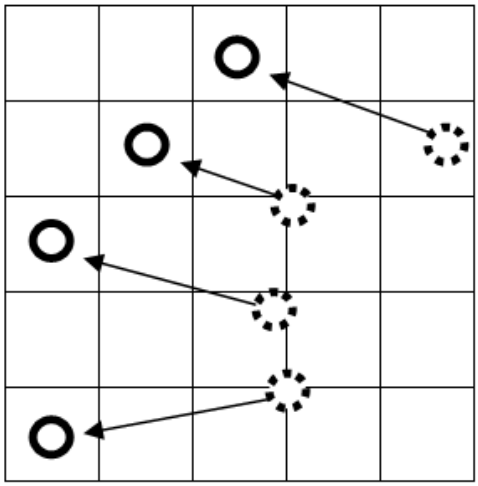

Figure 11: Tracing backwards to the source of the density

In figure 11, the solid circles are thought of as the destination and the dashed circles the source. To determine what densities are arriving at this location, examine the velocity field and determine, based on its magnitude, where the densities would have to originate from. Then,

simply take a weighted average of the four cells the densities will be arriving from to calculate the new density at the destination. Once this is done the cell states are updated with their new densities and the process repeated. Figure 12 shows the movement of densities through a fixed vector field.

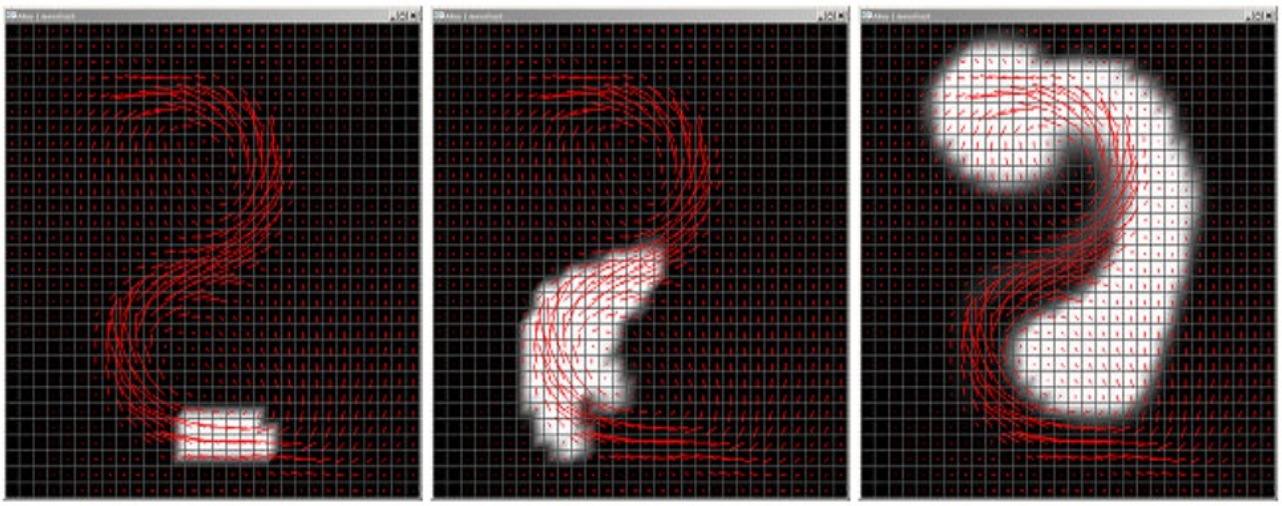

Figure 12: Moving densities through a fixed velocity field (used with permission)[5]

At this point the density values are updated and the density solver is complete.

The velocity solver has several additional steps to resolve the evolution of the velocity fields. The first steps make use of the *diffusion* and *advection* functions. What is nice about this algorithm is that the functions are identical when diffusing or advecting the densities as it is for the velocities. As previously mentioned the velocities are stored in their component form; therefore, when they are diffused they are calculated separately. The next step is the projection stage which is unique to the velocity solver.

During the calculations of the previous steps the results were rarely mass conserving; an important characteristic to maintain realism and stability in the model. Therefore the purpose of the projection step is to help conserve the mass and add some desired visual effects: swirls and eddies. In order for this to occur, the velocity field is defined as the sum of a mass conserving

field and a gradient field. To get the mass conserving field, the gradient field is subtracted from the current velocity field. The gradient field is calculated using a linear Poisson system. The projection step is repeated twice to help maintain accuracy after the advection step.

Finally, the behavior between the fluids and boundaries must be defined. For this thesis, the *no-slip condition* is chosen to model the fluid boundary interactions. The theory of the *no-slip condition* states that the fluid velocity is always zero at the boundary-fluid interface [22]. For instance, at the surface of the interaction between cell **A** (boundary) and cell **B** a fluid, in figure 13, the velocities in these cells need to average to zero. The velocities are never actually averaged; instead, the velocity for cell **A** becomes the negation of cell **B**. This way, the two cells averaged the result would be zero.

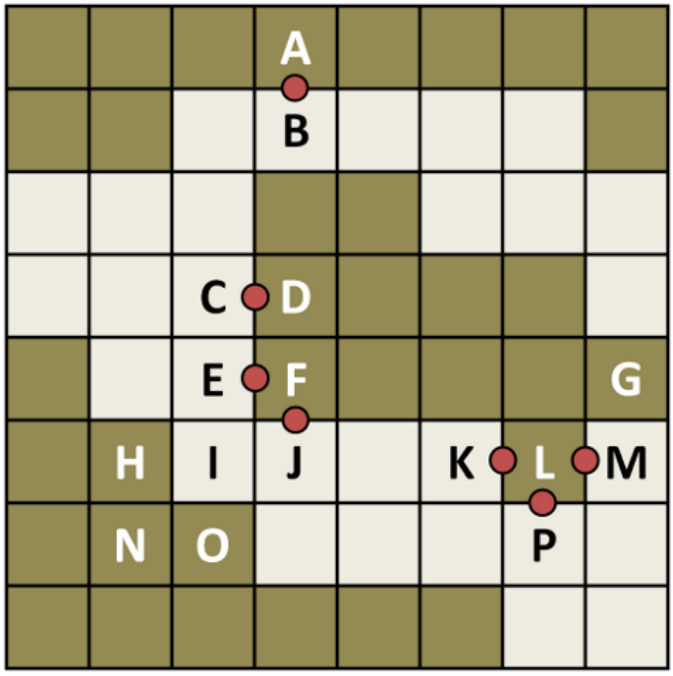

Figure 13: Model of Boundary Fluid Interactions with Boundaries Represented as dark cells and Fluids as light cells

The interactions between cells **A** and **B** and between cells **C** & **D** are straightforward since the interaction is between one fluid cell and one boundary cell. The interaction between

cells **E**, **J** and **F** is more difficult. The value for cell **F** becomes equal to the average of the negation of cells **E** and **J**. The worst situation is when there is an interaction between three fluid cells and a single boundary cell, as seen between **K**, **L**, **M** and **P**. However, such a situation can be avoided by not allowing boundaries or passage ways to be one cell wide.

4.2 Diffusion Function

The diffusion function is responsible for calculating the natural flow of the particles regardless of the forces exerted by the velocity fields. This natural flow, or diffusion, is represented in the Navier-Stokes equations (presented in equations 1 and 2 of section 2.1) as the radiation term. In addition to resolving the radiation of the densities, it is also responsible for resolving the radiation of the velocity field.

To calculate the new state value for the cell we must first determine two factors: the particles leaving the cell and the particles entering the cell from the immediately adjacent cells, as seen in figure 14. Therefore the new cell value will be equal to the previous state value minus the densities leaving to the surrounding cells plus the particles entering the cell, as seen in equation 7, which is derived by taking the existing densities and adding and subtracting the densities that are leaving and entering the cell space respectively.

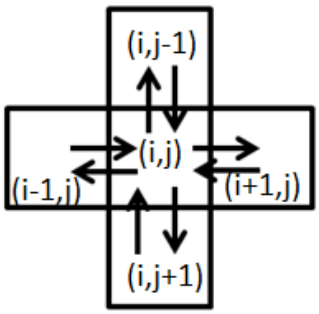

Figure 14: Diffusion Directions

$$D_{NEW}(i,j) = D(i,j) - D_{Leaving}(i,j) + D_{Entering}(i \pm 1, j \pm 1)$$

Equation 7: Calculating the new density values

In other words the total change in density would be equal to: d(i-1,j)+d(i+1,j)+d(i,j-1)+d(i,j+1)-4*d(i,j), the sum of the densities from the four adjacent cells, minus the densities leaving to the adjacent cells. A further adjustment must be made to incorporate the rate at which the densities diffuse. By multiplying the values by a factor of a, we can then adjust the rate of diffusion. Therefore, to calculate the new densities is as simple as: d'(i,j) – a*[d(i-1,j)+d(i+1,j)+d(i,j-1)+d(i,j+1)-4*d(i,j)]. This simple method, however, can lead to some instability for larger diffusion rates. Therefore, a more stable approach is to use a Gauss-Seidel relaxation to solve for the linear system [5] and the result equation 6. This is an iterative process and must be repeated until a stable solution has been acquired. The rate at which the densities radiate between cells is referred to as the viscosity and is incorporated into equation 6 as the value for **a**. By incorporating the viscosity into the equation, it is possible to simulate particles with different behaviors. A low viscosity would cause the densities to have very little diffusion, similar to a liquid; while a high viscosity would rapidly radiate to the surrounding cells and take the shape of the container, which is similar to the behavior of a gas. This equation ensures that densities are always travelling from a high concentration to a lower concentration.

The implementation of equation 7 with the Cell-DEVS formalism is straightforward. The *x(i,j)* term is replaced with the port for which the state value is calculated. For example when the values stored within the *diffusion* port are being diffused, the *x(i,j)* term is replaced by (0,0)~diffusion, similarly the *x(i±1,j±1)* terms are replaced by the corresponding neighbors' *diffusion* port values. Finally, for this example, the *x'(i,j)* is replaced with the state value stored in the *value* port.

The following is an example of the execution of the diffusion function in a zero velocity field. The viscosity had a value of 0.05 and the initial density field was a block of four by four with densities equal to one.

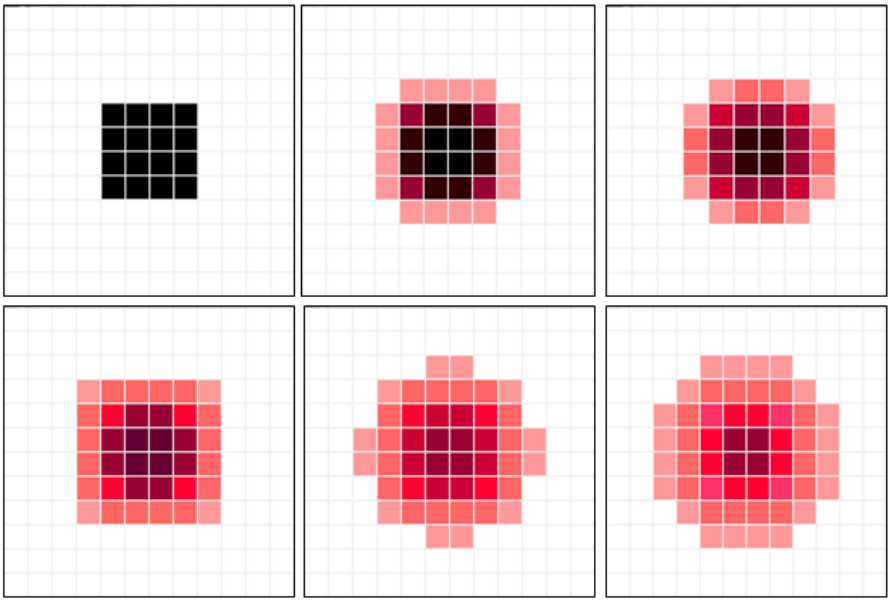

Figure 25: Diffusing densities over 25 iterations with a viscosity of 0.05

Figure 15 shows how the diffusion function is produces the desired results. With no external forces acting upon the density "cloud" the natural radiation causes the densities to spread out evenly to the surrounding cells. Another important feature to note is that there is no change in the overall mass of the system, ensuring that the diffusion function is mass conserving. For example the following are the values for the first and second frame shown in figure 14:

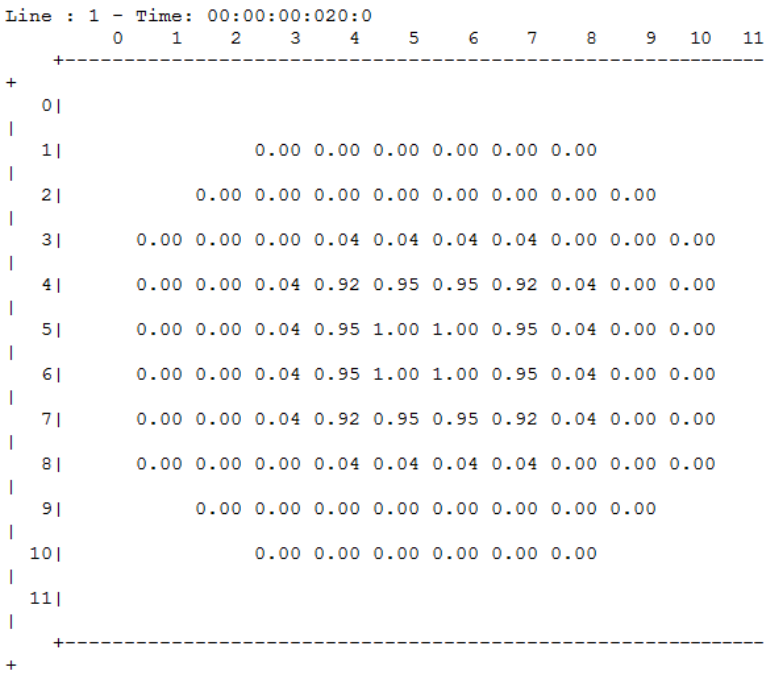

Figure 16: Values for the 1st frame of the diffusion function

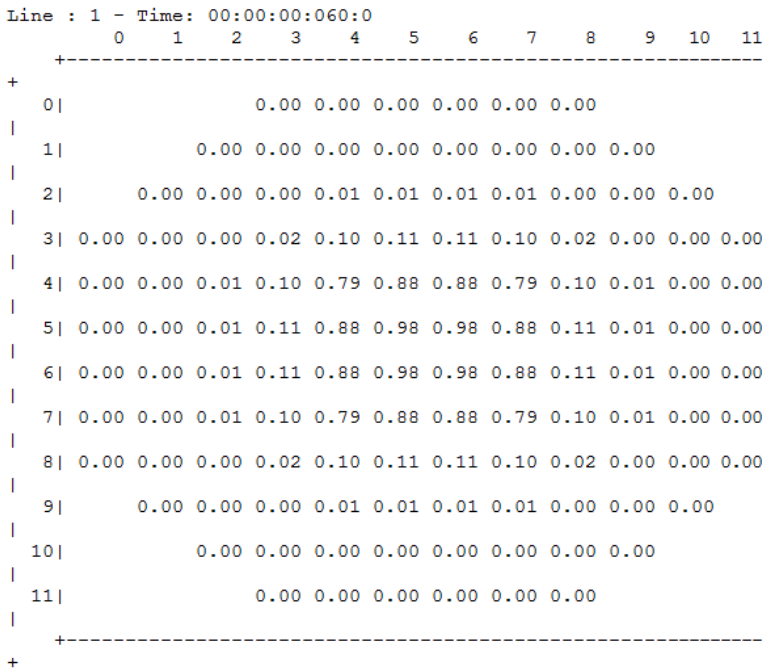

Figure 17: Values for the 16th frame of the diffusion function

The initial values used for this simulation was a 4 by 4 square with values of 1, which means the mass of the system is equal to 16. If we were to find the total mass of the first frame,

figure 16, it would be as follows: 4(1.00) + 8(0.95) + 4(0.92) + 16(0.04), for a total of 15.92. The total mass for the 5th frame, figure 16, would be found as follows: 4(0.98) + 8(0.88) + 4(0.79) + 8(0.11) + 8(0.1) + 4(0.02) + 12(0.01), for a total of 16. The slight discrepancy, 0.5%, from the initial values to the first frame is most likely due to rounding the results. It is important to note, that this rounding occurs after the simulation is complete, the values are stored in their entirety throughout the simulation. This small discrepancy is gone for the results of the 16th frame.

4.3 Advection Function

As previously mentioned in section 4.1, the advection function's purpose is to "move" the densities and velocity fields. This movement generated is caused by either the forces acting upon the density field from the velocity fields, or, in the case of the velocity solver, the momentum of the velocity fields. The velocity field is composed of velocity vectors which are stored in component form, hereafter referred to as velocity component vectors where *u* represents the horizontal component and *v* represents the vertical component. In other words, the advection function resolves the last term of the Navier-Stokes equations, section 2.1 equations 1 and 2, which is the forces applied to the system. While there are many methods for which the forces could be interpreted to resulting movements, the method used in this thesis is very stable and works with the Cell-DEVS formalism.

The approach to moving densities presented in this thesis is to determine the densities entering the cell instead of where the densities currently in the cell will end up. The method of tracing the origins of the densities is described in section 4.1. To implement this process we must first ensure that the origin of the densities lie within the defined neighborhood.

One approach would be to create a suitably large neighborhood that would ensure that, regardless of the magnitude and direction of the forces generated by the velocity field (hereafter referred to as velocity vectors), the densities would originate from within the neighborhood boundaries. This is however, is an impractical approach. The neighborhood would have to be extremely large to ensure the vectors did not exceed their boundaries and even then, the possibility does exist that with the addition of external forces, they might. Instead, if we can ensure that the velocity vectors will remain within a set range we can define a neighborhood with 100% certainty that the displacements will not exceed the neighborhood boundaries. In our model, the maximum absolute value of the magnitudes for the velocities is set to be one, therefore the neighborhood was defined as the 8 cells surrounding the cell being computed (called a Moore's Neighborhood). The magnitude of the velocity component vectors was limited to one for several reasons. First, as mentioned in section 2.1, if the modeler is not careful, large movements can cause instability in the model. The backwards tracing method is supposed to lead to a more stable model. Our method has no chance of becoming unstable since there is no risk of movements becoming too large. Second, as discussed later, larger velocities would require a larger neighborhood; this would result in the need for additional cases and overall increase the computational effort.

The probability that the location the densities originated from lies at the cell center is rare. For this reason, the densities that will be "moved" to the new cell location will most likely come from 1 or more cells. Therefore, the new density state value is calculated as a weighted average of the four closest cells to the origin location. For example, let us assume the velocity component vectors at the current cell are u = 0.6 and v = 0.4. The density values are represented by the *d'* array.

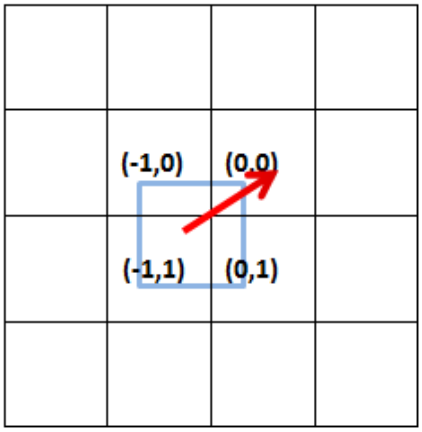

Figure 18: Results of the advection calculation

By drawing a square with the same dimensions as a cell, centered about the origin of the velocity vector, it is clear that the densities must come from all four cells. The amount of the densities coming from each cell is proportional to the area of each cell enclosed by the square. The following equation is used to determine these amounts for this case:

$$D(0,0) = \left[u \times \left(v \times d'(-1,0) + (1-v) \times d'(-1,1) \right) + (1-u) \times \left(v \times d'(0,0) + (1-v) \times d'(0,1) \right) \right]$$

Equation 8: Weighted averages for new density

Using the values previously described the results would be as follows: 36% of the density originates from the cell location (-1,1), while 24% from both (-1,0) and (0,1) cells and the remaining 16% comes from (0,1) cell; this totals to 100% therefore the formula is mass conserving. These values correspond to the theoretical values from figure 17.

As previously mentioned, the magnitudes for the velocity component vectors were restricted to fall between 1 and -1. These forces would translate to the maximum distance traveled to the cell being 1. This means there are four different combinations for which the velocity components could be. Therefore, four versions of equation 8 must be defined to

represent the possible outcomes. Figure 19 shows these four possible cases and the regions that they cover.

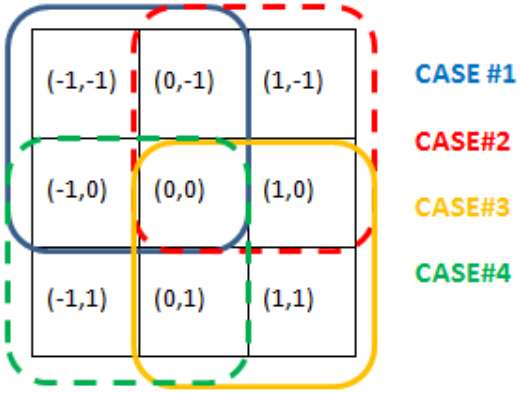

Figure 19: Break down of the possible sources of the density for the advection step

The first case is for when the velocity component vectors are between $0 \leq u \geq 1$ and $0 \leq v \geq 1$. The resulting equation is shown in equation 9.

$$D(0,0) = \left[u \times \left(v \times d'(-1,-1) + (1-v) \times d'(-1,0) \right) + (1-u) \times \left(v \times d'(0,-1) + (1-v) \times d'(0,0) \right) \right]$$

Equation 9: Weighted Average for new density Case #1

The second case is for when the velocity vectors are between; $-1 < u > 0$ and $0 < v > 1$. The resulting equation is shown in equation 10.

$$D(0,0) = \left[|u| \times \left(v \times d'(0,-1) + (1-v) \times d'(0,0) \right) + (1-|u|) \times \left(v \times d'(1,-1) + (1-v) \times d'(1,0) \right) \right]$$

Equation 10: Weighted Average for new density Case #2

The third case is for when the velocity vectors are between; $-1 \leq u \geq 0$ and $-1 \leq v \geq 0$.. The resulting equation is shown in equation 11.

$$D(0,0) = \left[|u| \times \left(|v| \times d'(0,0) + (1 - |v|) \times d'(0,1) \right) + (1 - |u|) \times \left(|v| \times d'(1,0) + (1 - |v|) \times d'(1,1) \right) \right]$$

Equation 11: Weighted Average for new density Case #3

The final case, case 4, is for when the velocity vectors are between; $0 < u > 1$ and $-1 < v > 0$. The resulting equation is shown in equation 12.

$$D(0,0) = \left[u \times \left(|v| \times d'(-1,1) + (1 - |v|) \times d'(-1,0) \right) + (1 - u) \times \left(|v| \times d'(0,1) + (1 - |v|) \times d'(0,0) \right) \right]$$

Equation 12: Weighted Average for new density Case #4

The advection function is therefore represented by these 4 equations. These same equations are used for moving the velocity vectors as well, with the only difference being that instead of density values being used, the magnitudes of the velocity vectors are being averaged.

4.4 Projection Function

The projection function is responsible for calculating the first term of the Navier-Stokes equation, as seen in section 2.1 equations 1. The main role of that term is to ensure the solution to the equation remains mass conserving. The projection function is the most complex function of the entire model and therefore, the information generated during the execution is stored in two separate ports, while the remaining information from the function is stored within the two vector component ports. These two parts are hereafter referred to as the *div* and *p* functions.

The *div* function is responsible for creating a gradient map. A gradient map shows changes in the velocity fields, with small values representing a uniform field with little variation and large values representing extreme fluctuations in the velocity field. To ensure the system remains stable, i.e. mass is not lost or created, we want to ensure that situations do not arise were the velocity vectors all converge to a point or diverge from a point, as seen in figure 19.

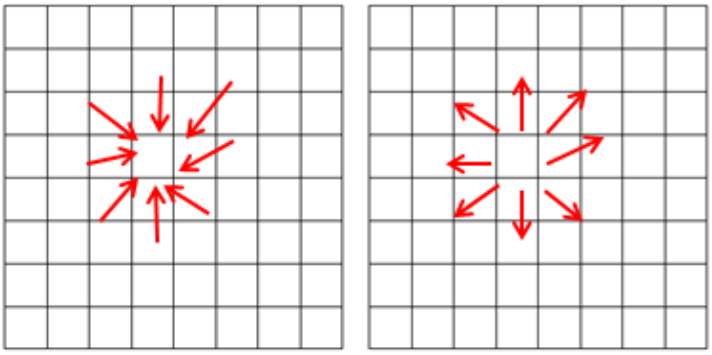

Figure 20: Convergent and Divergent Velocity Fields

Figure 20 shows two situations we do not wish to occur, hereafter referred to as convergence and divergence. With all the velocities pointing to one cell, it could cause instability. For example, while the densities can theoretically exceed a value of 1 without causing instability this may not be a desired outcome. More importantly though is this may cause the magnitude of the velocity component vectors to exceed 1 at this location and that is not allowed. Similarly, with the divergent case, were all the velocities are leading away from the cell, the density at the center cell would follow each velocity vector without being divided amongst all of the vectors. This would result in more mass leaving the cell then was actually present in the cell, once again leading to instability. The equation for generating the gradient map is as follow, equation 13:

$$div[i,j] = -0.5 * \frac{1}{h} * \big(u(i+1,j) - u(i-1,j) + v(i,j+1) - v(i,j-1) \big)$$

Equation 13: Generating the gradient field

As we can see, the gradient field is calculated as the sum of the differences of the vertical and horizontal neighbors. Therefore, a small gradient value will occur when the change in values of the neighborhood is small. Large gradient values will occur when the values of the velocities make extreme changes but still maintained the same sign. The largest, and worst case, is when

the values are opposite signs on either side of the cell. This would be what we see happening in figure 19.

Once we have obtained the gradient field, the *p* function is executed. Similar to the *diffusion* function it helps average the values out, reducing large gradient changes. The equation for the *p* function is as follows, equation 14:

$$p(i,j) = \frac{[\, div(i,j) + p(i-1,j) + p(i+1,j) + p(i,j-1) + p(i,j+1)]}{4}$$

Equation 14: averaging the gradient field

This equation is run for several iterations.

The final step is to subtract the gradient field from the velocity field. The result will be a more stable field, hopefully void of any situations such as those shown in figure 20. The following two equations, equations 15 and 16, represent the gradient field being subtracted from the velocity field.

$$u(i,j) = (\, u'(i,j) - 0.5 * h * \big(\, p(i+1,j) - p(i-1,j)\big)$$

Equation 15: Subtracting the gradient field from the horizontal field

$$v(i,j) = (\, v'(i,j) - 0.5 * h * \big(\, p(i,j+1) - p(i,j-1)\big)$$

Equation 16: Subtracting the gradient field from the vertical field

Again, for small changes in the velocity field, the *p* values will be equal and therefore the final velocity field will not be changed. Even for large differences that occur between velocities of the same sign, the *p* value will have been averaged such that the final difference in the *p* values will not be that large, and will not affect the final field significantly. Only during situations, such as those presented in figure 20, will the final velocity field be changed much.

This completes the projection function. As seen in section 4.1 figure 10, the *projection* function is repeated twice for the velocity solver. This is to ensure that changes made to the

velocity field during the *diffusion* or *advection* functions do not cause any convergent or divergent behaviors in the velocity field. For example, if two "fronts" are about to collide, the *diffusion* process could bring them close enough together such that a case similar to figure 19 exists. This would cause problems to occur during the *advection* process. Therefore the *projection* function is used to help negate this. However, this would only effect the leading edges of these "fronts" and once again when advected the fronts may come near enough that they are convergent. During the *diffusion* process these fronts would be averaged and the result could be that they cancel each other out causing a dead space. This is not the desired effect; therefore, we call the *projection* function once more to ensure this does not occur.

4.5 Boundary Function

The purpose of the *boundary* function is to define the behavior of fluid-boundary interactions. To implement the *no-slip condition* for the boundaries in DEVS, let us look at figure 13, section 4.1. The no-*slip condition* states that the velocity should average to zero along the boundaries. To implement this, we added an additional function to both solvers called the *boundary* function. The main goal is to ensure the system remains mass conserving and provides the behavior of the fluid-boundary interactions.

The most important role of the *boundary* function is to control the behavior of the velocity fields around the boundaries. As previously stated, the *no-slip condition* states that the average of the velocity field should be zero along the edge of the boundaries. What is nice about this model is that the velocities are stored in component form. This mean the only boundaries that are of interest to the *u* vectors are the surfaces that run perpendicular to the vectors, in this case vertical boundaries, while the *v* vectors look at the horizontal boundaries. When running the

boundary function for the vector ports, trace along the boundary and set the vector ports for those boundary cells to be equal to the negative of the neighboring non-boundary cell's corresponding vector port. That the average of the two values the result would be zero. This zeroing of the velocity field will stop the densities from interacting with the boundaries. If we ensure that no boundaries are a single cell wide, we do not have to worry about a situation arising similar to that of L in figure13. By ensuring that boundaries are more than one cell wide, we reduce the loss that would be generated from having to average the values of more than two cells. When a boundary is in contact with only one non-boundary cell, it is equal and opposite to the state value of the non-boundary cell. When in contact with two non-boundary cells, it needs to assume a state value that is equal and opposite to the average of the adjacent cell's state values. This introduces the possibility for some loss of mass to occur, however it is minimal. It is important to note that when in contact with more than two non-boundary cells, as seen in case **L**, it is possible for **M** and **K** to cancel each other when averaged. This would introduce the possibility for major loss of mass to occur. A similar situation occurs when channels are one cell wide, and therefore, neither channels nor boundaries can be one cell wide.

As previously mentioned, the other role of the boundary function is to ensure the system remain mass conserving, i.e. that the presence of boundaries does not negatively impact the system. Therefore, the *boundary* function is integrated when the other functions are called. For example, to ensure no mass is lost during the diffusion of the densities, the boundary cells assume a value that is equal to the average of the surrounding non-boundary cells, as seen in equation 17.

$$D_{Boundary}(i,j) = \frac{(Sum\ of\ Non\ Boundary\ Cells)}{\#\ of\ Non\ Boundary\ Cells}$$

Equation 17: Boundary Equation

Other times, the applications of the *boundary* function is less complex and exist solely to ensure nothing "wanders into" a cell that has been defined as a boundary.

4.6 Implementing the Algorithm as a Cell-DEVS in CD++

As previously stated, our model consists of two main parts, each consisting of several functions. When expressing the algorithms functions as a Cell-DEVS rule in the CD++ language several considerations are necessary. First, the amount of information being generated and stored during the execution of this model is too great to be stored as a single state value within the cell, as in a traditional Cellular Automaton. To deal with the different pieces of information, we used different input/output ports to transmit the necessary state values. The first one is the *value* port. Its purpose is to store the results of each iteration of the algorithm generated during the simulation of the model: the *time*, *location* and *magnitude* of the densities (and, if needed, the location of the *boundaries*). Then, the *diffusion* port is responsible for executing the diffusion function in the density solver. These results are then advected when the *value* port updates the state values at the end of the iteration. These two ports can be thought of as the *density solver* function presented in section 4.1.

The next two ports we use are the *u* and *v* ports. Their main responsibility is to store the component vectors of the velocity field, which are used to apply the forces during the *advection* function. These two ports are also responsible for executing the *advection* function for the velocity solver, as well as the last stage of the *projection* function.

The next two are the *p* and *div* ports. Their responsibility is to receive information related to the two parts of the *projection* function, and activate the computation of such function described earlier in section 4.4. The final port is the *boundary* port. The role of this port is to

simply store the location of the boundaries. These locations are necessary when the *boundary* function is called during the execution of the other functions.

Table 1: Timing Breakdown for a Single Iteration

	Density Solver Ports		Velocity Solver Ports			
Time	Value	Diffusion	U	V	P	Div
1	Boundary	Boundary	Boundary	Boundary	Reset	Reset
2		Diffusion				Div
3		.				Boundary
4		.			P	
5		.			.	
6		.			.	
7		.			P	
8		.	Projection	Projection	Reset	Reset
9		.	Boundary	Boundary		
10		.	Advection	Advection		
11		.	Boundary	Boundary		
12		.				Div
13		.				Boundary
14		.			P	
15		.			.	
16		.			.	
17		.			P	
18		Diffusion	Projection	Projection	Boundary	
19	Boundary	Boundary	Boundary	Boundary		
20	Advection	Reset	Diffusion	Diffusion	Reset	Reset

The second consideration that must be made to implement the model is the timing. Because the algorithm used is complex and relies on functions being implemented in series, it is not possible to complete the algorithm in a single time step. Secondly, to improve the resolution it was decided that several functions would be iterated over several time steps. For this model, it was decided that one complete iteration/frame would last 20 time steps. The specific number of time steps per iteration can vary depending on many factors, such as desired resolution and efficiency of the model. For this thesis, the time duration of an iteration was set at 20, since beyond this point there was little change in the resolution for the *diffusion* step and it allowed

sufficient time for all the other steps to run till completion, Furthermore, an even number that was also a multiple of 100 helped in programming the model. Table 1 breaks down which functions are being executed in which ports during specific time steps. Note the *boundary* port is not included since it does not implement any functions, since its sole purpose is to store the border locations.

Some important things to note from table 1 are: First, the *advection* function at time step 20 for the *value* port signifies the end of the iteration. However, to increase efficiency, the next iteration has already started for *u* and *v* ports. Because the cells are updated asynchronously, changing the state values of these ports at the same time they are being used will not affect the results. Next, the *p* function is constantly running; however, with the resets and its dependency on their being values stored in the *div* port, the cell state values remain zero.

As discussed in section 2.3 the state values of a cell are determined by the local computing function. The local computing function consists of a **PostCondition, Delay** and **PreCondition.** As with all programming languages, the methods used vary depending on the individual programmers' style. For this model, it was decided that the algorithm would be implemented as a single rule, and therefore the **PreCondition** would is set to be always true. To do this, it is necessary to have a method of determining which time step it is. This was done by using a built-in variable of the simulator called the *time* and the *remainder* function (remainder of the long division function). By using the *remainder* function in conjunction with *if* statements we are able to define the behaviors for all the functions in section 5, based on the input ports data, for any given time step within a single rule.

The first two ports transmit all the information generated by the density solver function. As discussed earlier, the *value* port is responsible for transmitting the results of the *advection*

function, as described in section 4.3. The following is the rule that defines the behavior of the functions activated when an input is received in the *value* port:

```
~value :=
// Initialization
if( time = 0 ,
      if( cellpos(1) < 20 and cellpos(1) > 5 and cellpos(0) < 6,
            0.8, 0)  ,
//Boundary Function
if( (0,0)~boundary = 2, -1,
//Advection Function
if( remainder(time,20) = 0,
      if( ( (0,0)~u >= 0 and (0,0)~u <= 1 ),
      //This can be either be case #1 or #4
            if( ( (0,0)~v >= 0 and (0,0)~v <= 1 ),
            //If yes then Case #1
                  ((0,0)~u*((0,0)~v*(-1,-1)~diffusion + (1 -
                  (0,0)~v)*(-1,0)~ diffusion) + (1 -
                  (0,0)~u)*((0,0)~v*(0,-1)~ diffusion+ (1 -
                  (0,0)~v)*(0,0)~ diffusion) ) ,
            //If not Case#1 than it must be Case#4
                  ((0,0)~u*( abs((0,0)~v)*(-1,1)~ diffusion+  1 -
                  abs((0,0)~v))*(-1,0)~ diffusion) + (1 -
                  (0,0)~u)*(abs((0,0)~v)*(0,1)~ diffusion+ (1 -
                  abs((0,0)~v))*(0,0)~ diffusion) ) ),
      if( ( (0,0)~v >= -1 and (0,0)~v <= 0 ),
      // Since u is negative it must be either Case#2 or #3
            // if v is negative than it is Case #3
            (abs((0,0)~u)*( abs((0,0)~v)*(0,0)~ diffusion+ (1 -
            abs((0,0)~v))*(0,1)~ diffusion) + (1 -
            abs((0,0)~u))*(abs((0,0)~v)*(1,0)~ diffusion+ (1 -
            abs((0,0)~v))*(1,1)~ diffusion) ) ,
            // if v is positive than it is Case #2
            (abs((0,0)~u)*( abs((0,0)~v)*(0,-1)~ diffusion+ (1 -
            abs((0,0)~v))*(0,0)~ diffusion) + (1 -
            abs((0,0)~u))*(abs((0,0)~v)*(1,-1)~ diffusion+ (1 -
            abs((0,0)~v))*(1,0)~ diffusion) )  )))
            )
,(0,0)~value )));
```

From the above code, three terms will affect the state value stored in the *value* port. These functions will occur either on the very first time step, or at the 20th time step. The first *if* statement represents the initialization of the port where it may be necessary to provide some initial values. Therefore, the first **PostCondtion** is that if *time* is equal to zero, i.e. the first time step, then the density values will be initialized as a 5 by 13 rectangle with values of 0.8. These

initial values can be adjusted to whatever is desired. The next *if* statement is for the *boundary* function and will occur on the 20th time step. As mentioned in section 4.5 the *boundary* function varies for each function it is integrated with. When integrated with the *advection* function, as seen here, it simply ensures that no densities are moved into cells that are defined as boundaries, in this case, cells whose boundary port is equal to 2. For the purpose of visualizing the results it was decided that the boundaries could be included in the *value* port and would appear as a negative value. The final series of *if* statements are for the implementation of the *advection* function, as described in section 4.3. This will occur on the 20th time step of each iteration of the model. The four cases, as presented in section 4.3 equations 9 through 12, are represented here where (0,0)~value is used to represent the *D(0,0)* and the *d'* terms are represented by the (i,j)~diffusion port values. It is worth noting that the cases do not appear in the same order as they were presented in section 4.3. This was to optimize the efficiency of the algorithm by limiting the number of *if* statements. Instead of using four longer *if* statements for each case, three shorter statements were used that grouped together like elements. For example, the long version of the *if* statement would be as follows:

```
if(   ( (0,0)~u >= 0 and (0,0)~u <= 1 and (0,0)~v >= 0 and (0,0)~v <= 1  )
```

This would have to be repeated four times, one for each case. Instead, by sharing common elements, as we did here, we are able to represent all four cases with three shorter statements. While this may not result in a significant difference in the execution time, over many iterations the small improvements in time should improve the efficiency slightly.

The final port used to store the information generated by the density solver function is the *diffusion* port. As discussed earlier, it is used to store the results from the *diffusion* function, and

provide the state values used during the *advection* function, as was just described. The implementation of the functions triggered by data received in this port is as follows:

```
~diffusion:=
if( (time = 0 or remainder(time,20) = 0), 0 ,
if( (0,0)~boundary = 2 ,
//Boundary Function
      ((0,1)~diffusion*( 1 - ((0,1)~boundary)/2) + (0,-1)~diffusion*( 1
      - ((0,-1)~boundary)/2) + (1,0)~diffusion*( 1 -
      ((1,0)~boundary)/2) + (-1,0)~diffusion*( 1 - ((-
      1,0)~boundary)/2)) / (  ((0,1)~boundary)/2 + ((0,-1)~boundary)/2
      + ((1,0)~boundary)/2 + ((-1,0)~boundary)/2 ),
//Diffusion Function
      ((0,0)~value + 0.05*( (0,-1)~diffusion + (0,1)~diffusion + (-
      1,0)~diffusion +(1,0)~diffusion))/( 1.2)
      ));
```

To put the code in words, when the remainder of time is zero the state values are reset to zero, otherwise both the boundary and diffusion functions are run. The *diffusion* port can be thought of as a temporary variable and therefore must be reset so previous values to not contaminate the calculations performed during the next iteration of the model. If the cell is a border cell, then it runs the boundary function; otherwise, it executes the diffusion function. In this case the boundary function's purpose is to conserve mass and try to minimalize the loss of mass to the border cells during the diffusion process. Equation 17 from section 4.5 describes how this is done.

The implementation of this equation could be done in one of two ways. First, it could be set up as a series of cases, similar to what was done for the *advection* function; however, with four neighboring cells there are 16 possible cases. Instead, the equation was implemented in such a way that it would cover all 16 cases. As seen above, the equation is such that it is the sum of the four adjacent neighbors divided by a maximum of four. However, any cell that is defined as a boundary automatically zeroes the term so it will not affect the overall sum. To reflect this in the

divisor term, it is the sum of the number of cells which are not defined as boundaries; where once again if a cell is defined as a boundary it zeroes that term.

The implementation of the *diffusion* function is straightforward. Equation 8, from section 4.1, is implemented, where the *x* variable is replaced with the value from the *value* port and the *x'* terms are the values from the corresponding cell's *diffusion* ports. To improve the resolution of the results it was decided that the *diffusion* function would be iterated 18 times. When we look at table 1, we can see that the *projection* function required the most time steps. As later discussed, it was decided that the equations used to generate the values stored in the *p* port would be iterated several times as well. Therefore, the minimum iteration length would be 16 time frames; this is the minimum number of frames the model can be completed in. This would mean the *diffusion* equations would be iterated 14 times; 14 iterations would ensure the *diffusion* function is complete in time for the *advection* function to occur. While 14 iterations of the equation would be sufficient for low viscosities, to ensure good resolution for higher viscosities the number of iterations was increased to 18; 18 iterations was chosen to ensure that even with high viscosities a decent resolution was obtained. Furthermore, by choosing 18 iterations as the duration for the diffusion equation, this brought the total number of time steps for one iteration of the entire model to 20, which is an easier number to handle during the coding process.

Similarly, the velocity solver function requires four ports to store the results from the execution. These ports are the *u*, *v*, *div* and *p* ports. As we can see from table 1, equations used to determine the **PostCondition** for the velocity component vector ports is similar. These ports are responsible for storing the results from the *diffusion, advection* and *projection* function. These functions are applied to the component vectors of the velocity field, therefore they are stored in

component form were *u* is the horizontal vector and *v* is the vertical vector. The implementation of the algorithm for determining the state values of these two ports is as follows:

```
~u:=
if( (remainder(time,20) = 1 or remainder(time,20) = 9 or
remainder(time,20) = 11 or remainder(time,20) = 19),
//Boundary Function
      if( (0,0)~boundary = 2,
            if( (1,0)~boundary !=2,
            -1*(1,0)~u, if( (-1,0)~boundary != 2, -1*(-1,0)~u,0)),
      (0,0)~u),
//Projection Function
if( (remainder(time,20) = 8 or remainder(time,20) = 18),
      (0,0)~u - (0.5)*(441)*( (1,0)~p - (-1,0)~p ),
//Advection Function
if( remainder(time,20) = 10,
      if( ( (0,0)~u >= 0 and (0,0)~u <= 1 ),
            if( ( (0,0)~v >= 0 and (0,0)~v <= 1 ),
                  ((0,0)~u*((0,0)~v*(-1,-1)~u + (1 - (0,0)~v)*(-1,0)~u)
                  + (1 - (0,0)~u)*((0,0)~v*(0,-1)~u  + (1 -
                  (0,0)~v)*(0,0)~u ) ),
            ((0,0)~u*( abs((0,0)~v)*(-1,1)~u + (1 - abs((0,0)~v))*(-
            1,0)~u) + (1 - (0,0)~u)*(abs((0,0)~v)*(0,1)~u + (1 -
            abs((0,0)~v))*(0,0)~u) )) ,
      if( ( (0,0)~v >= -1 and (0,0)~v <= 0 ),
            (abs((0,0)~u)*( abs((0,0)~v)*(0,0)~u + (1 -
            abs((0,0)~v))*(0,1)~u) + (1 -
            abs((0,0)~u))*(abs((0,0)~v)*(1,0)~u  + (1 -
            abs((0,0)~v))*(1,1)~u )),
      (abs((0,0)~u)*( abs((0,0)~v)*(0,-1)~u + (1 -
      abs((0,0)~v))*(0,0)~u) + (1 - abs((0,0)~u))*(abs((0,0)~v)*(1,-
      1)~u + (1 - abs((0,0)~v))*(1,0)~u)) )),
//Diffusion Function
if( remainder(time,20) = 0 and time != 0,
      ((0,0)~u + 0.05*( (0,-1)~u + (0,1)~u + (-1,0)~u +(1,0)~u))/1.2 ,
if( time = 0,
      uniform(0.7,0.8) , (0,0)~u
)))));
```

First the code for defining the **PostCondtion** for the *v* port is similar to the above code. The code presented here can be separated by *if* statements. The first *if* states that during the 1st, 9th, 11th and 19th time step the boundary function is used to determine the new state values. If this is not true then the next statement is examined. If the time step is equal to 8 or 18 then the *projection* functions dictates the new state values. Else if not true, then the time step is equal to 10 and the *advection* function will update the velocity field ports. If this is not true then the time

step is equal to 20 and the diffusion function is executed to determine the new state values. The final term is only true during the very first time step and it is here that the initial port values for the velocity field can be described.

The implementation of the *diffusion* function, as described in section 4.2, is similar to how it was implemented for the density solver; however, instead of the particles being diffused the magnitudes of the velocity component vectors are being diffused. Also, it was decided that this function would only be iterated once, therefore x is represented by the current magnitude of the component and the x' are represented by the magnitudes of the respective neighboring cells. As outlined in table 1, the *diffusion* occurs at the 20^{th} time step. The implementation of the *advection* function, as described in section 4.3, is similar to how it was implemented for the density solver. Again, the difference here is that the component vectors are being moved. The cases remain the same, as outlined by equations 9-12 from section 4.3, with the cell's new state value representing $D(i,j)$ and the $d'(i,j)$ terms coming from the respective neighbors of the same port. As outlined by table 1, this occurs during the 10^{th} time step. During the 8^{th} and 18^{th} time step, the last section of the *projection* function occurs. As described in section 4.4 equations 15 and 16, this is when the gradient field is subtracted from the velocity field. Again, since the velocity field is stored in component form, equation 15 is used to solve for the new horizontal component vectors and equation 16 is used to solve for the new vertical component vectors. Finally, during the 1^{st}, 9^{th}, 11^{th} and 19^{th} time steps the *boundary* function is implemented. As, outlined in section 4.5, this is done by having the boundary cells assume a state value equal to the negative average of the surrounding cells. An advantage to storing the velocity field in component form is that for situations such as **F** from figure 13, section 4.1, we are able to store the negative horizontal component from **E** in the u port while the negative vertical component

from **J** is stored in the *v* port. This makes it so the values do not need to be averaged, like it does when the *boundary* function is called for the density solver, which improves the overall results.

The next two ports are responsible for storing part of the results from the *projection* function. Specifically, the *div* port stores the results generated by equation 13 from section 4.5 and the *p* port stores the results generated by equation 14 from section 4.5. As described in section 4.5, the *div* port is responsible for storing the gradient field generated during the *projection* function. The implementation of *div* port is as follows:

```
~div:=
//Reset
if( remainder(time,20) = 0 or remainder(time,20) = 8, 0,
// Projection Function
if( remainder(time,20) = 2 or remainder(time,20) = 12,
      -0.5*(1/441)*( (1,0)~u - (-1,0)~u + (0,1)~v - (0,-1)~v),
//Boundary Function
if((remainder(time,20) = 3 or remainder(time,20) = 13) and
(0,0)~boundary = 2,
      ((0,1)~div + (0,-1)~div + (1,0)~div + (-1,0)~div)/4
,(0,0)~div
)));
```

Simply put, during the 1st and 8th time step the gradient field is reset to zero. During the 2nd and 12th time step the gradient field is calculated and during the 3rd and 13th time step the *boundary* function applied. Here the *boundary* function sets the state value of cells defined as boundaries to be equal to the average of the surrounding cells.

The final port used to store the information generated from the *projection* function is the *p* port. This port is responsible for storing the results generated by equation 14 from section 4.5. In other words, this port stores the averaged gradient field. The results are later accessed when the gradient field is subtracted from the velocity field, as previously discussed. The implementation of equation is as follows:

```
~p:=
//Reset
if( remainder(time,20) = 0 or remainder(time,20) = 8, 0,
```

```
// Projection Function
((0,0)~div + (-1,0)~p + (1,0)~p + (0,-1)~p + (0,1)~p)/4);
```

The averaged gradient field is reset on the 1st and 8th time step; otherwise the gradient field is being averaged. At first glance this does not agree with what is presented in table 1, however, after the 1st and 8th time step the field is reset to zero in both the *p* and *div* ports. Therefore without any values stored in either port the field will remain zero. Once the *div* port has the values stored for the new gradient field, the *p* port will begin to store the averaged gradient field. This means, in essence, that the *p* port is only storing new results during the 4th through 7th and 14th through 17th time steps, which is in agreement with table 1. The averaging of the gradient field was chosen to be iterated more than once to improve the resolution. The specific number of iterations was to round out the total length of the algorithm to 20 time steps.

The previous four ports store all the information that is generated by the velocity solver function. Together with the first two ports, we have completed both functions.

The final port to define is the *boundary* port. Oddly enough it is one of the two ports in which the *boundary* function is not called. Instead the role of this port is to define a cell as either a boundary or non-boundary cell. This is done by setting the corresponding state value to 2 or 0 respectively. It is also in this port that the design/shape of the boundaries can be implemented. For example, the following code implements a hollow square of 50 by 50 cells:

```
~boundary :=
if( time = 0 and
      (cellpos(1) = 0 or cellpos(1) = 49 or cellpos(0) = 0 or
      cellpos(0) = 49),
 2, (0,0)~boundary);
```

A more complex example of the definition of the boundaries is shown in the following:

```
~boundary := if( time = 0 and
(cellpos(1) = 0 or cellpos(1) = 24 or (
      (cellpos(1) < 3 or cellpos(1) > 21) and cellpos(0) > 15 and
      cellpos(0) < 60   )
or (
```

```
      (cellpos(1) < 5 or cellpos(1) > 19) and cellpos(0) > 20 and
      cellpos(0) < 55   )
or (
      (cellpos(1) < 7 or cellpos(1) > 17) and cellpos(0) > 25 and
      cellpos(0) < 50   )
or (
      (cellpos(1) < 9 or cellpos(1) > 15) and cellpos(0) > 30 and
      cellpos(0) < 45   ))
```

This example provides the boundaries that appear in the simulation presented in section 5.2

All together this is the implementation of the rule that provides the behavior of the algorithms presented in section 4.1.

5. Simulations of CFD Model

As stated in section 2, the goal of this thesis was to create a CFD that was able to provide realistic results. Therefore, in this chapter we will be demonstrating that our model is able to do so. This will be done by presenting different test results for each of the algorithms used in developing the model, and providing a real-world application for the model. We will look at the effect the narrowing of the coronary arteries due to plaque buildup will affect the flow of blood to the heart muscles. The simulations presented in this section were executed remotely on the RISE server [28] using CD++ v 2.0 [29]. The results are then downloaded and visualized using the visualization engines of the CD++ Toolkit.

5.1 Testing the Model

Before running complex simulations, it was necessary to see if all the components and functions of the model were functioning properly. The first test scenario, as seen in figure 20, was to test the *diffusion* and *advection* functions. It was executed with a grid space of 75 cells by 75 cells and was exposed to a uniform velocity field indicated by the red arrow.

The exact values were: u between 0.8 and 1.0 and v between 0.1 and 0.4. The simulation was run for 175 iterations. The boundary conditions were set to *wrapped* to simulate a looped grid. The viscosity for both the densities and the velocities were set to 0.05.

From figure 20, we can see that the *diffusion* functioned performed as expected. The density foci expanded outwards uniformly with a relatively low viscosity. There appears to be no loss of mass either. Near the end of the simulation, the size of the density foci began to remain constant. At first glance, it would appear that the densities were no longer diffusing. This, however, is not the case; instead, the magnitudes of the densities near the edge of the foci were becoming too small, i.e. less than 0.1, and therefore cannot be seen in the visualization. This is

an important feature of this model, which demonstrates how its flexibility and how the model can be implemented for a wide range of applications. Often when the densities become too low, we no longer wish to keep track of them (smoke, for example), and therefore by not storing these low values we can help reduce the computational load. However, in some scenarios the densities might only exist in small quantities, like drugs circulating in the blood stream. If this is the case, the threshold can be adjusted in the model and even the smallest of amounts can be tracked.

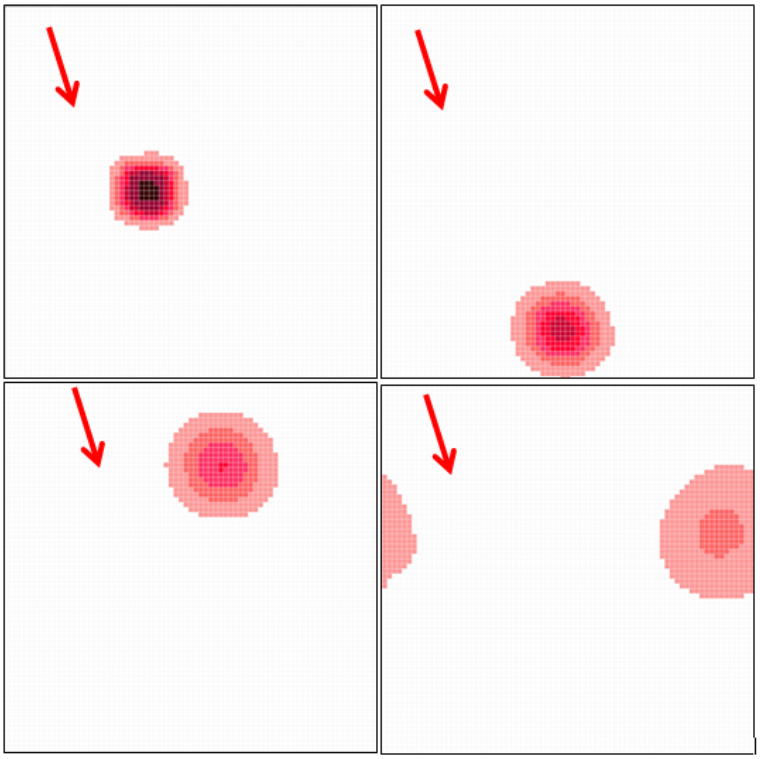

Figure 21: Snapshots of density cloud being advected with the frames progressing from left to right, top to bottom

The *advection* function worked well in the uniform velocity field. With the average component vectors being 0.9[S] and 0.25[E], the resulting displacement should be 0.934 units [S15°E], as seen in Figure 22. This magnitude and direction clearly match the results seen in figure 20.

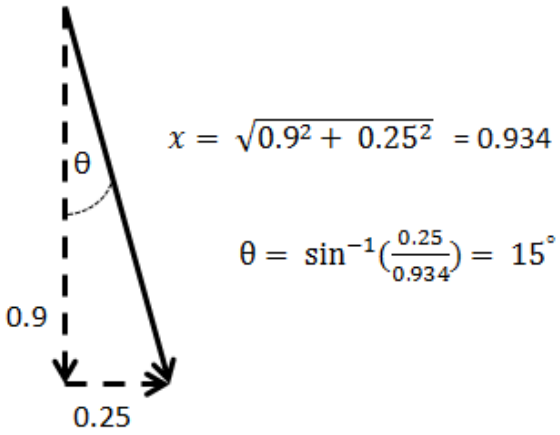

Figure 22: Resulting Velocity vector

In addition, the density foci made approximately two rotations through the cell space. With a height of 75 cells that mean the foci travelled approximately 150 cells vertically during the simulation. The simulation lasted 175 frames which means the foci moved an average of 0.86 cells per frame which corresponds to having a vertical velocity of 0.9. Overall the two functions worked seamlessly together and provided great results with a strong resolution.

The next simulation that was executed was to test the velocity field and to see how it would behave under non-uniform velocity conditions. The parameters were the same as the first test with only the initial velocities being different. Figure 23 shows the initial values for the simulation.

The goal was to determine the behavior of the evolution of the velocity field when two distinct velocity fronts came into contact. As before, the first regions average velocity was 0.934 [S15°E] while the second regions average velocity was 0.814[E11°N].

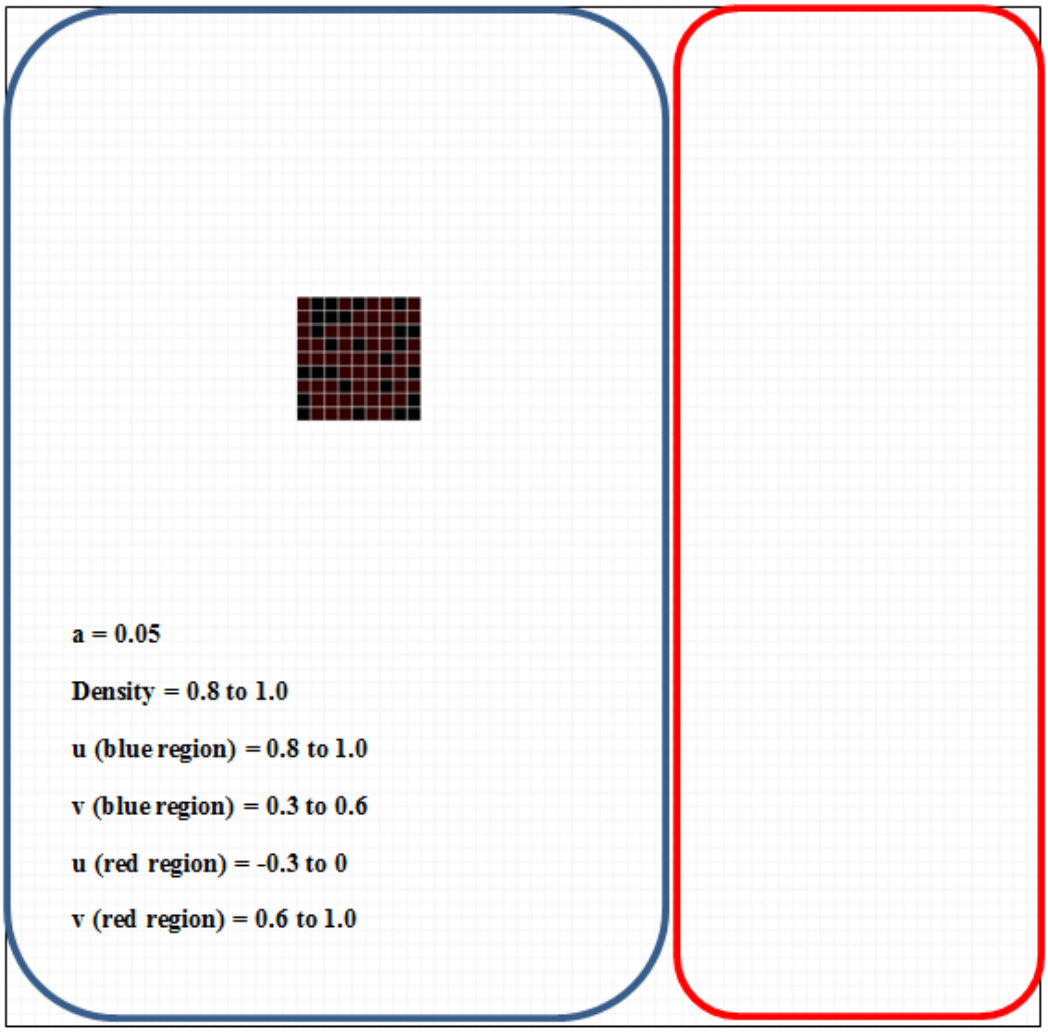

Figure 23: Initialization for non-uniform velocity field

The results of this simulation can be seen in figure 24. The red arrow in the top right corners represents the approximate forces acting on the density cloud.

In this test, we were interested more in how the densities moved when under non-uniform conditions, in order to provide insight into how the velocity fields evolved. In the first frame, the forces were as expected, and the advection path of the cloud was similar to the one seen in figure 20. By the second frame, the cloud would be entering the red section, as outlined in figure 23. Here again, the forces seemed to be as expected: the horizontal velocity increased and the large

downward force was slowed by the smaller upward force present in the second section. By the 4th frame, the cloud had passed through the region of horizontal velocity and returned to the region with the downward forces.

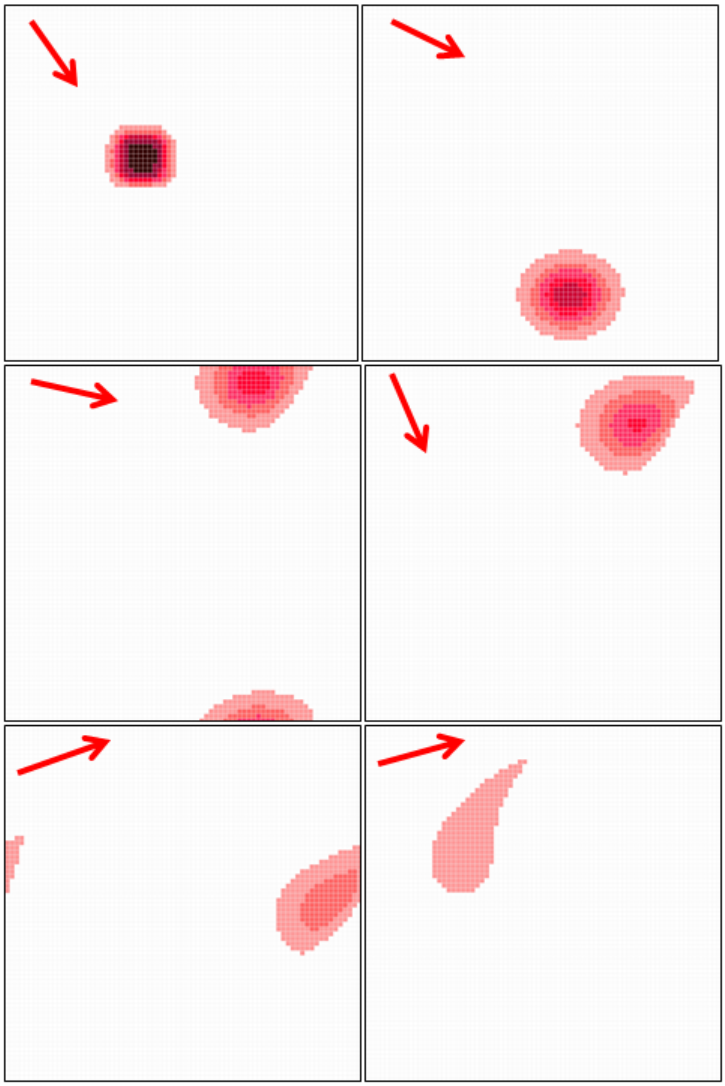

Figure 24: Testing for non-uniform velocity field with frames representing progression of time from left to right and top to bottom.

The most interesting were the last two frames, where a region of upward forces survived and was able to provide visible forces on the cloud. What is surprising about this is the time at which it occurred. This was near the very end of a long simulation and it was believed that the velocities would have approached a more uniform distribution of magnitudes. This clearly is not

the case, which means the projection function worked well at preserving these eddy-like flows. This is another good sign that the model is behaving as it should.

To investigate the behavior of the velocity field further, the simulation was executed again, however this time the values for the velocity component vectors were stored.

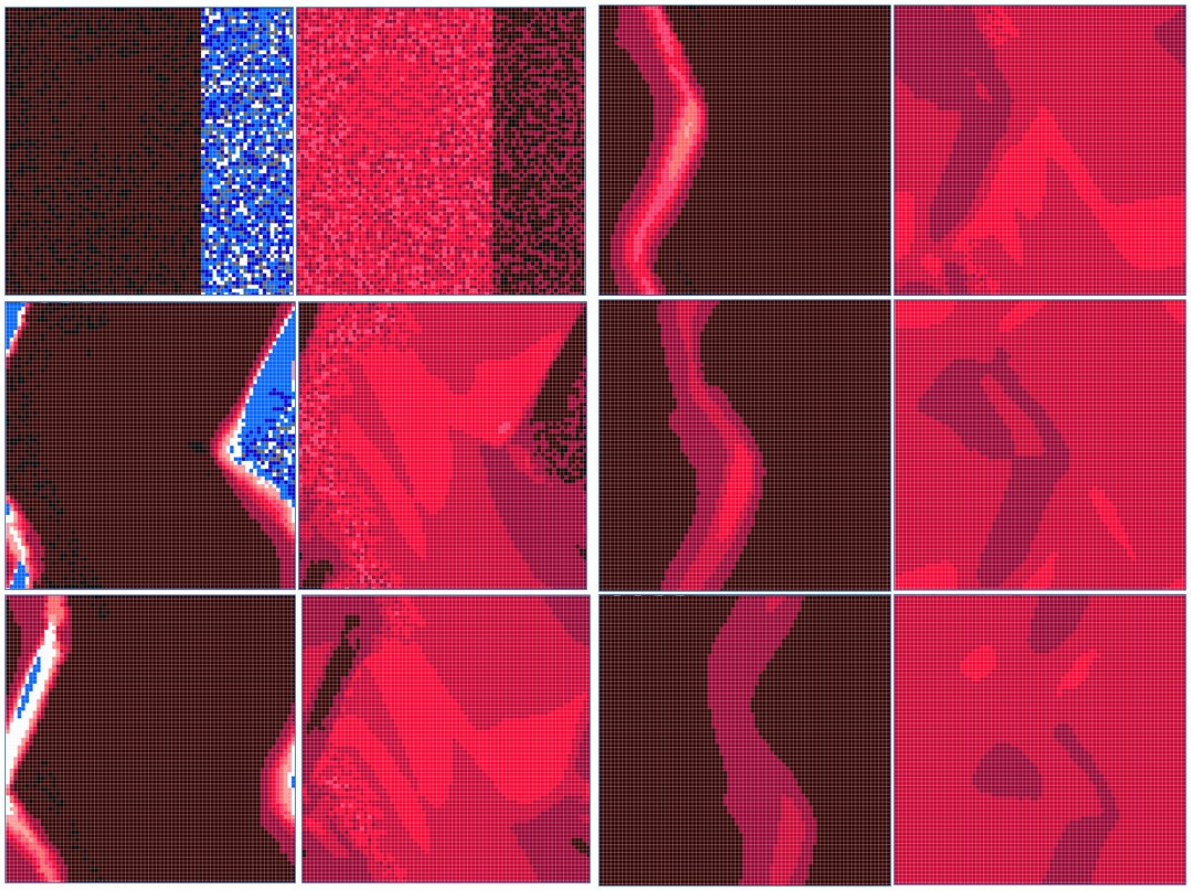

Figure 25: Results for velocity vectors u and v for non-uniform velocity field with frames representing progression of time top to bottom.

Figure 25 shows the results from the same test, but now displaying the values of the *u* and *v* ports. In figure 25, we get to see how the velocity fields interact. The first column is the result for the *u* port. What is interesting is that the negative velocity values lasted a long time with no external forces added. It was not until the 4th frame, 125 iterations later that the negative values disappeared due to diffusion. Even then a noticeable band of low velocities still remained.

Similarly, the *v* vectors had two separate zones where they differed in magnitude and not sign. Because the overall direction of the vectors was still the same, they seemed to diffuse together. Likewise, since regions of the high magnitude vectors were located inside the negative *u* zone, it did not maintain the solid band of high magnitudes like the *u* vectors did. Eventually, both fields would become uniform, since no external forces were added to the system. The results from figure 25 show that the *advection* and *diffusion* functions are functioning as they should for the velocity solver. A more chaotic field would be required to see the effect of the projection function as a visual. The mass conserving portion of the projection function clearly worked since none of the vectors exceeded their maximum size and the final field seems to be equal to the average of the initial field. As described in section 4.4, the projection function is used to resolve convergent and divergent behaviors. Since the vertical forces, which have opposite signs, are not coming into direct contact with each other (a convergent behavior), there is no crucial role for the *projection* function to play.

The next two simulations were used to test the introduction of boundaries and how the functions behaved with obstacles present. The scenarios were set up similar to a wind tunnel, where the velocity field was blowing in from one direction and a density cloud injected into the tunnel. The one key difference was that we still wished to see how the velocity field interacted with the boundaries as well as the densities, and therefore, instead of a continuous input of force on the velocity field, similar to a true wind tunnel, a single initiation of the velocity was used. Figures 26 and 27 shows the results for these tests with arrows used to represent the forces applied by the velocity fields.

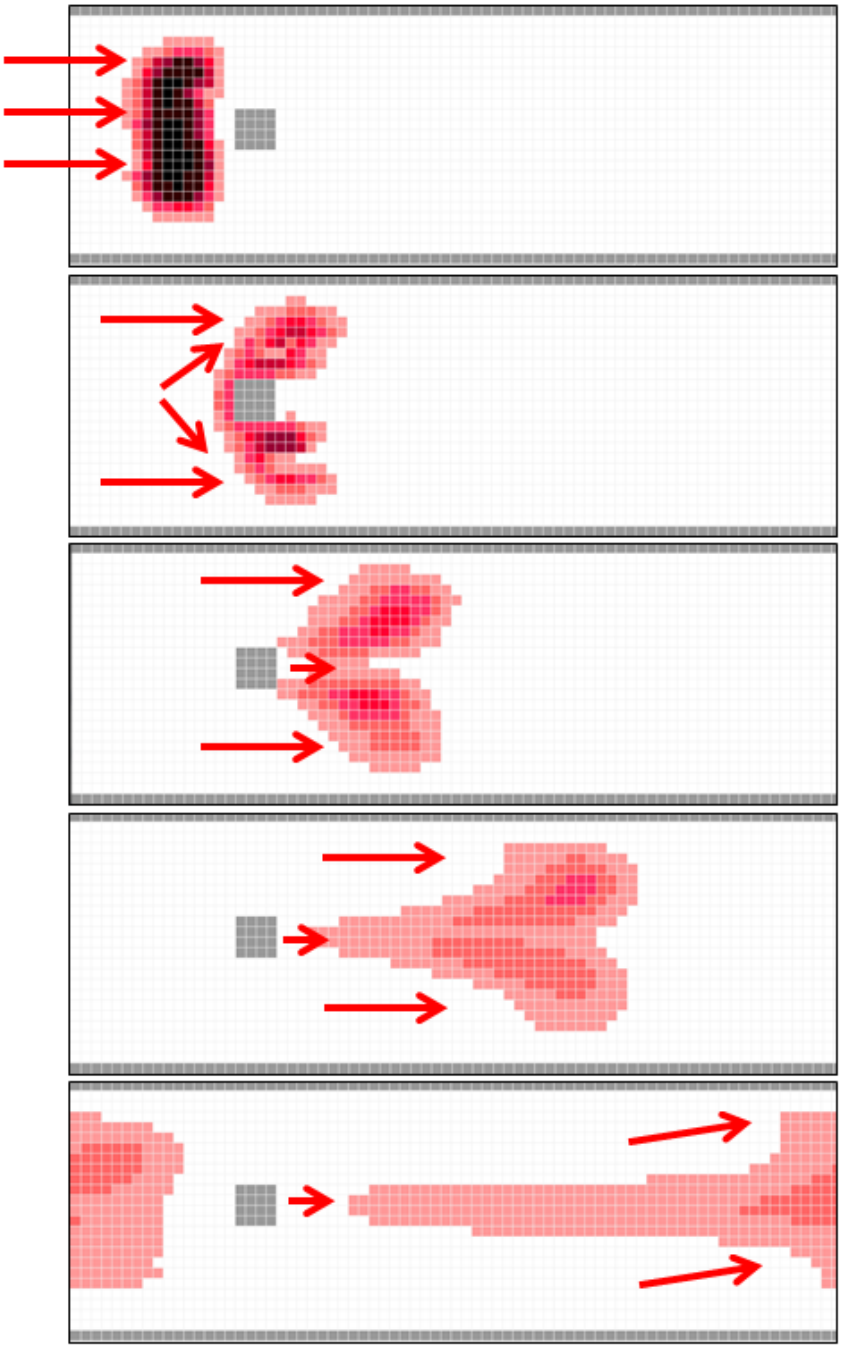

Figure 26: Diffusion of densities in a horizontal velocity field with barriers with time progression from top to bottom.

In figure 26, we can see a density focus encountering a fixed obstacle. With the viscosity set relatively low for both the velocity and the density (0.05), we can see that the focus splits into two distinct clouds that are then pulled back together, mostly by the velocity field. With the

viscosity of the velocity field being low, we see a space of zero velocity directly behind the obstacle, as we would expect. The following is the initial velocity field and the velocity field after 50 iterations.

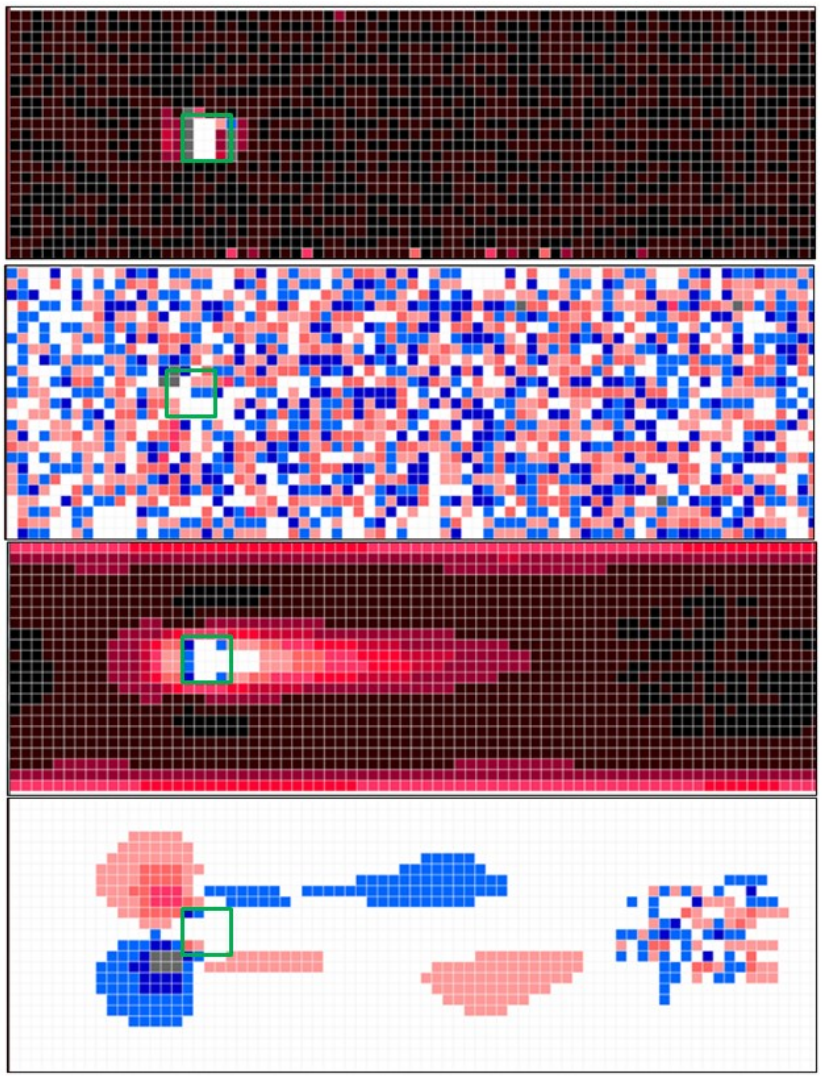

Figure 27: Velocity Component Vectors *u* and *v*: initial values and mid simulation values with green square representing the obstacle

The next simulation presented here used the exact same parameters as before, however, the obstacle was made larger and the velocities only were allowed flowing around in one path.

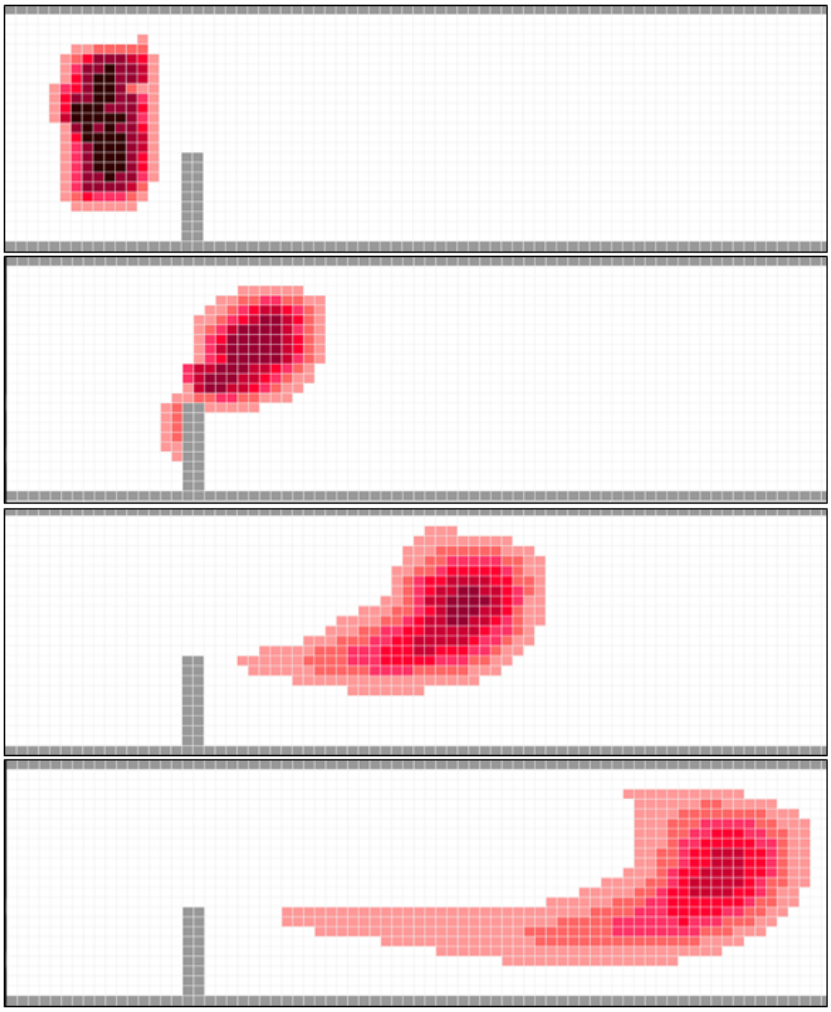

Figure 28: Diffusion of densities in a horizontal velocity field with barriers with time progression from top to bottom.

In figure 28, we see the focus encountering the obstacle. The majority of the density is redirected by the evolving velocity to flow around the obstacle and stay within a high velocity field. However, through diffusion and the momentum of the foci, some of the density entered the low velocity field caused by contact between the field and the obstacle. This can be seen as the long tail that extends from the foci. Again an area of null velocity occurred behind the obstacle, however, this time the obstacle was larger and the field could only flow around in one direction. Consequently, the null region extended further behind the obstacle.

As seen in figure 26, the velocity fields behaved as expected. The *no-slip* boundary condition stated that the velocity should be zero along the surfaces. What we expected to see was a slowing down of the velocities near the surfaces of boundaries and that is what we see in the results, best noticed in the second frame of figure 27. In the first scenario, the obstacle was small and the velocities were able to travel around either side. As we would expect, the velocities immediately behind the obstacle are zero and this null zone took the shape of a teardrop. However, when the density had passed the obstacle, some of it was "pulled" backward into this null zone in a similar fashion to an eddy current. This current eventually stabilized and the densities escaped. The second simulation involved the use of a larger obstacle, and it only allowed the velocities to flow around one side of it. As expected, the null zone behind the obstacle was larger and no eddy current was produced.

As previously discussed the purpose of this test was to examine the fluid-barrier interactions, and how they are resolved using the *no-slip* conditions. The first function to look at is the *diffusion* function, which is more obvious when used in the density solver, since the density focus spreads out over time. The diffusion of the velocities is more subtle. Without the diffusion of the velocity field, the null zone behind the obstacle would most likely be larger since there is no force pulling the velocities into this area of low pressure. The diffusion function is clearly the driving force that provides the force that draws the velocities into this null region.

The next function to look at is the *advection* function. Again, the most obvious application of the *advection* function that can be noted is the movement of the density foci. The forces are being correctly applied to the density field, since the resulting behavior is as expected. The more subtle expression of this function is in the velocity field. The initiation of both models was the same in that a uniform velocity field was generated with no null zones. The null zones

are created by the momentum of the velocity driving itself forward, and the obstacle preventing velocities from taking the place of the forces that are left. This creates a zone where the velocities are zero, which disappears further from the obstacle due to forces diffusing into the area. From the results generated, it is clear that the *advection* function is behaving as expected.

As previously described, the *projection* function was responsible for ensuring that the fields remain mass conserving and for adding visual effects. First, as can be seen in figure 26, the velocity vectors never exceed their allowed size. As we expected the velocities were slowed by the presence of barriers. More importantly though is how the *projection* function handled the collision between the velocity field and the obstacle. As described in section 4.5, the velocity value for the "boundary" cells should be equal to the negative of the neighboring cells, as seen in the first frames of figure 27. As described in section 4.4, this would cause a convergence in the velocity field, which causes instability. It is to handle these situations that this function exists. As seen in the second set of frames from figure 27, it handles this convergence of forces by taking the forces entering the area directly in front of the obstacle from the velocity field, and diverting the horizontal forces up and down. This is a perfect solution to the situation, and reflects the real-world behavior of such a fluid-boundary interaction. The final goal of the projection function is to create visual effects such as eddies. In the real world, forces would be drawn into the null region created behind an obstacle. As seen in figure 27, the *projection* function causes forces created to drive the densities into this space.

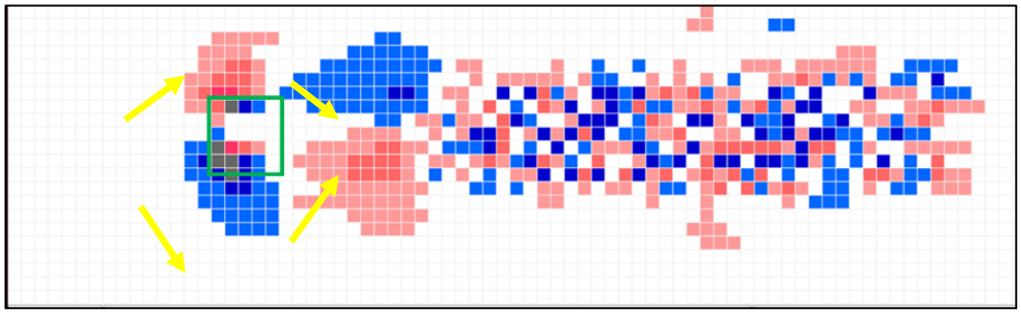

Figure 29: Vertical velocity component for first simulation: green square shows the location of the obstacle

Figure 29, shows the vertical forces acting on the densities during the second frame of figure 25. As we can see, upon initial contact with the obstacle the *projection* function creates a force that pushes the densities around the obstacles and pulls them back in behind the obstacle to the area of low pressure, as shown by the yellow arrows.

The results of our simulations showed that the individual functions are working exactly as expected. The integration of these functions built a model that is fully functional and accurately describes the mechanics of the fluid flow.

5.2 Coronary Artery Disease

Coronary Artery Disease (CAD) is the leading form of heart disease and the leading cause of heart attacks, resulting in the most deaths world-wide. CAD happens when plaque builds up on the artery walls. This accumulation of plaque hardens, and thus narrows the arteries. This narrowing restricts blood flow through the arteries, and since these arteries are supplying blood to your heart, the restriction of blood flow weakens it. The dangerous part of CAD is that typically patients suffering do not show any immediate signs or symptoms. Eventually, the myocardial cells will become ischemic from the lack of oxygen and potentially cause a heart attack [30].

To detect for CAD, doctors often perform an angiogram. In an angiogram, dye is injected into the coronary artery via a catheter and a rapid series of x-ray images are used to track the flow of the dye through the arteries and detect any narrowing or blockages [31].

The following simulation is an attempt to demonstrate how the narrowing of the arteries affects the blood flow. Several scenarios will be run: a control test blockage (0%), a minor blockage (17%), a medium blockage (35%), a major blockage (52%), and a late stage CAD blockage (70%). A single bolus of dye will be "injected" into the artery that is initialized with a uniform velocity field. The following are the results of these simulations:

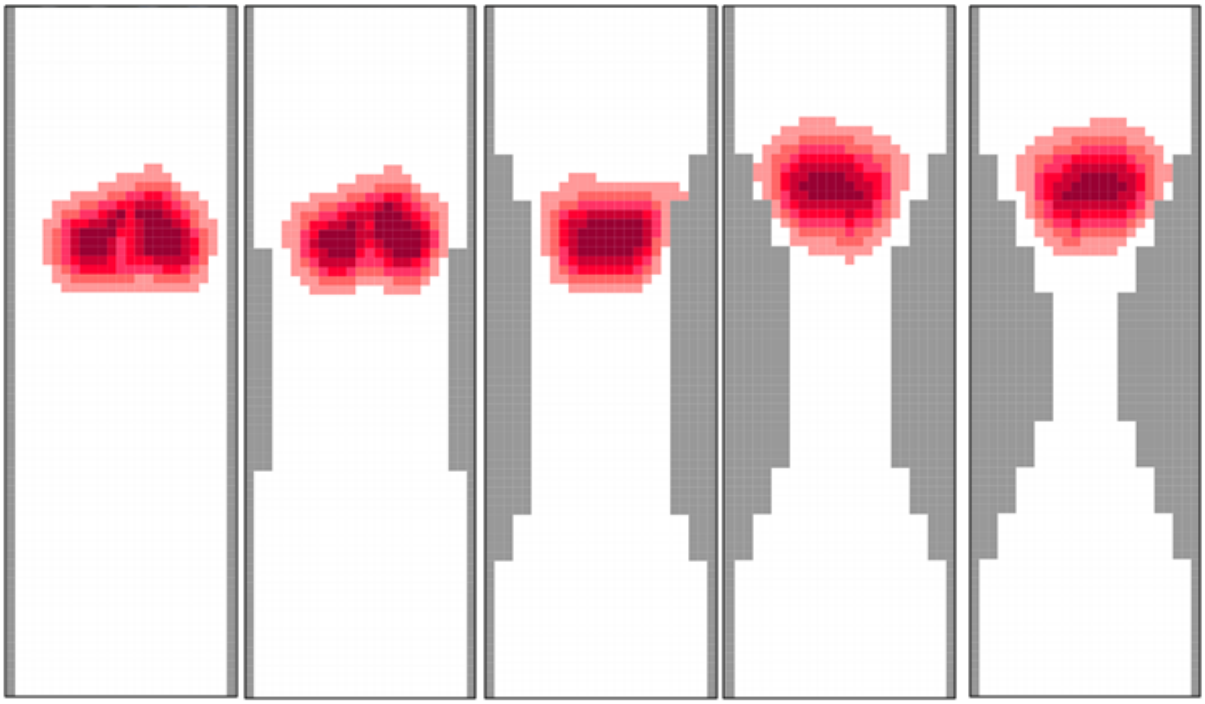

Figure 30: Simulation of CAD after 25 iterations: 0%, 17%, 35%, 53% and 70% blockage respectively

Figure 30 shows the results for all the scenarios after 25 iterations have passed. There is no significant difference between 0% and 17% blockage manly due to the size of the bolus and the size of the blockage. The 35% blockage shows the bolus being more concentrated and not being

allowed to diffuse as much, however there is still little effect on the velocity field. The 52% and 70% blockage show a decrease in the velocity field due to the narrowed arteries. Because of the narrowing of the channel width, the velocities are being slowed. This is resulting in a slower movement of the bolus in the two cases of severe blockage. As time progresses we should see a more significant slowing due to the blockage.

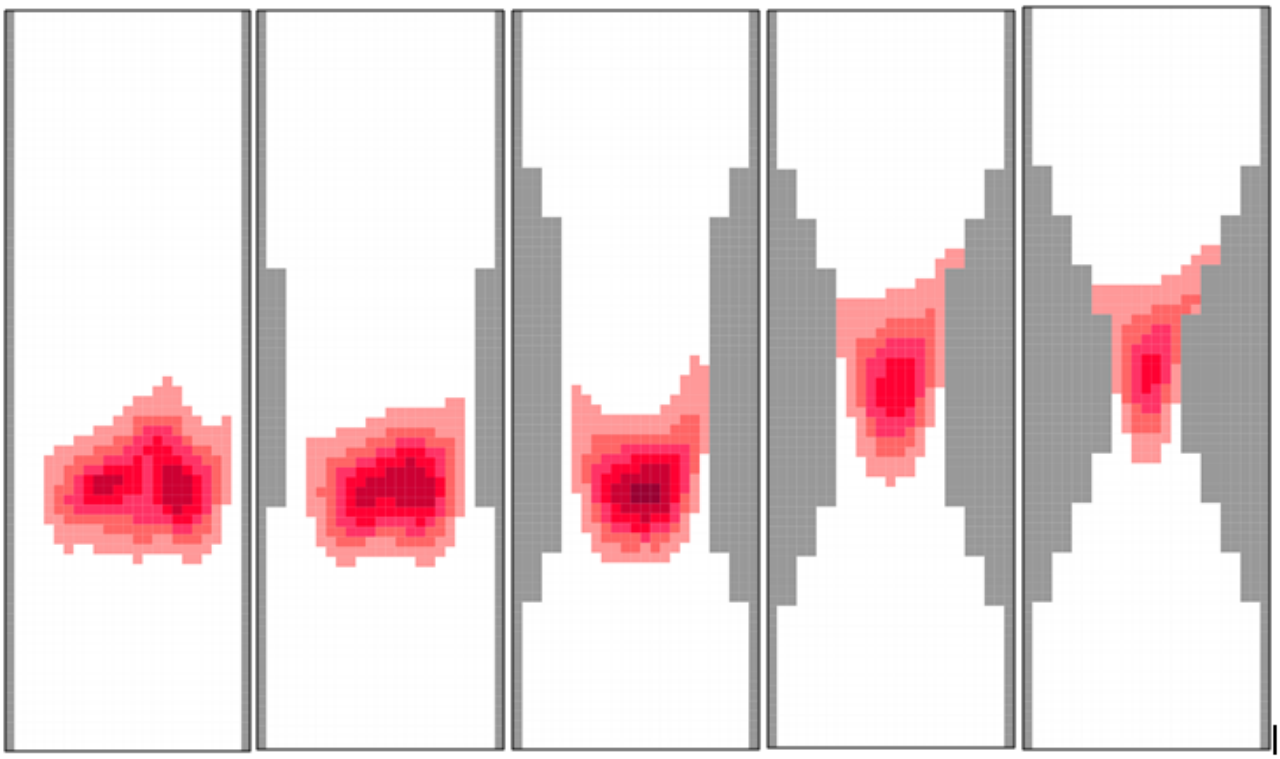

Figure 31: Simulation of CAD after 50 iterations: 0%, 17%, 35%, 53% and 70% blockage respectively

Figure 31 shows the results after 50 iterations have passed. Again there is no significant difference between the first three scenarios; however, the larger blockages are slowed significantly and have not passed through the blockage site yet. This is expected since the behavior of the fluid-boundary interactions is such that flow decreases near the boundary walls and with a narrow channel the flow is slowed across the width of it. Already we are starting to see the dangers of CAD. The blockage is causing a significant reduction in the speed at which

the “blood” is traveling thus hampering the hearts ability to deliver oxygen to the muscle tissue. The following shows the vertical velocities at this point in time for the 53% blockage.

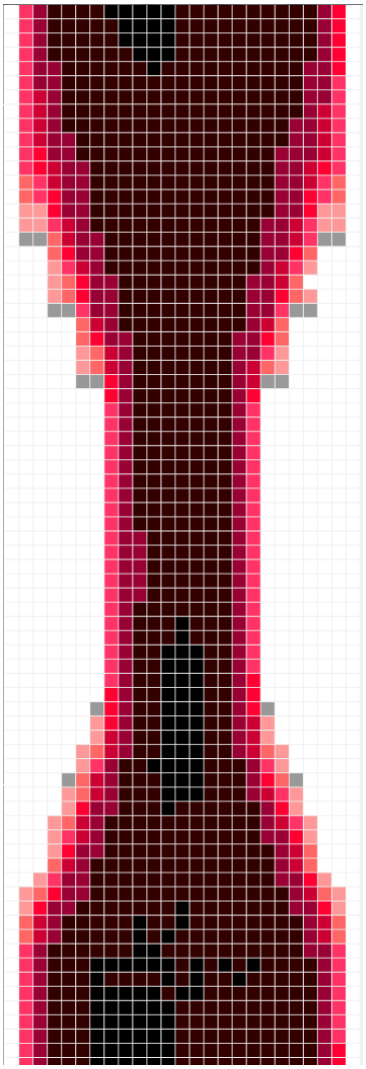

Figure 32: Vertical Velocity Field for 53% blockage after 50 iterations

As seen in figure 32, the narrowing of the artery/channel, has caused the velocity field to decrease.

The boundary conditions for these simulations were set to be *wrapped* which cause the flow to be continuous from top to bottom. This allowed for a longer section of artery with

multiple blockages to be represented with a smaller model. It is during the second pass of the bolus that we start to see the most significant results.

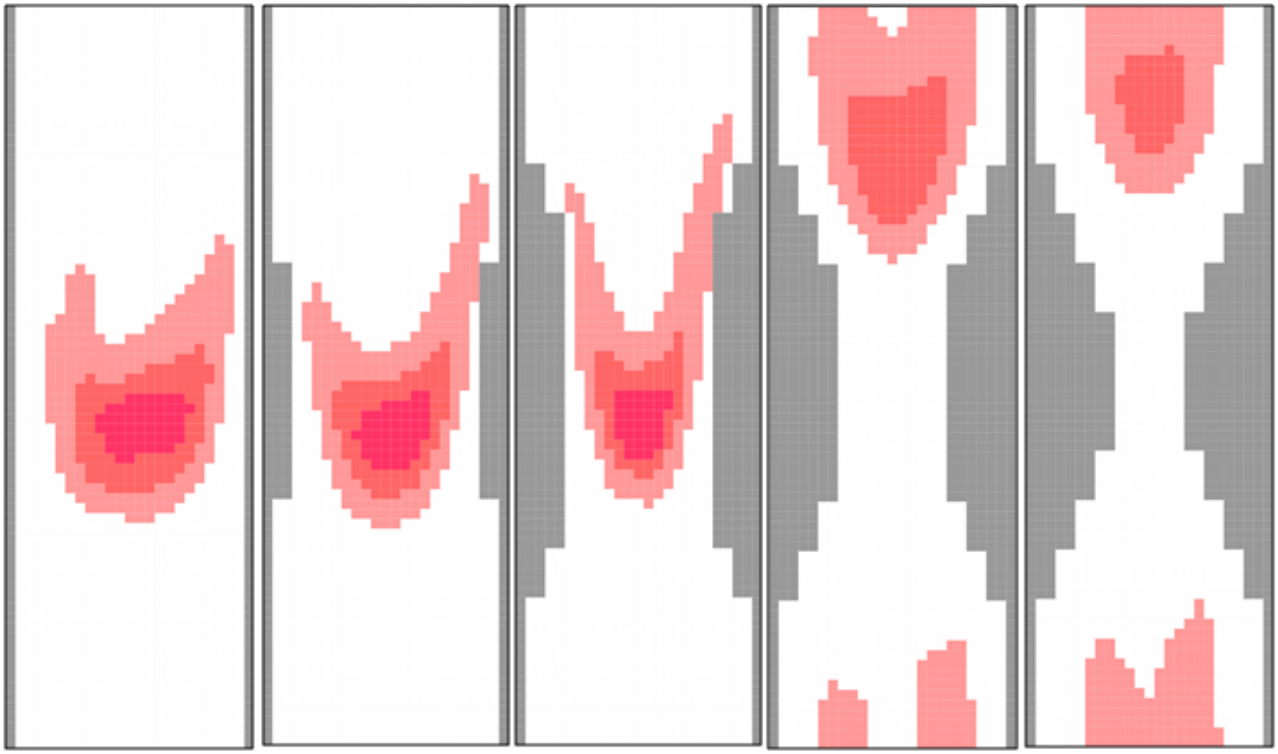

Figure 33: Simulation of CAD after 125 iterations: 0%, 17%, 35%, 53% and 70% blockage respectively

By the 100th iteration the first three scenarios are making their second pass through the blocked section of artery. The 35% blockage scenario is showing a decreased velocity field along the edges of the artery resulting in the bolus being longer. Also, the bolus has fallen behind from the control signifying a slight decrease in the flow rate. The last two scenarios show a large decrease in the flow rate and in the concentration of the bolus that has made it through. This demonstrates that at these levels of blockage we would most likely see a drop in the amount of oxygen being delivered to the nearby heart muscle which could result in a weakening of the heart

function. In figure 33 we see what would happen if there was a second blockage downstream of the first, of equal size.

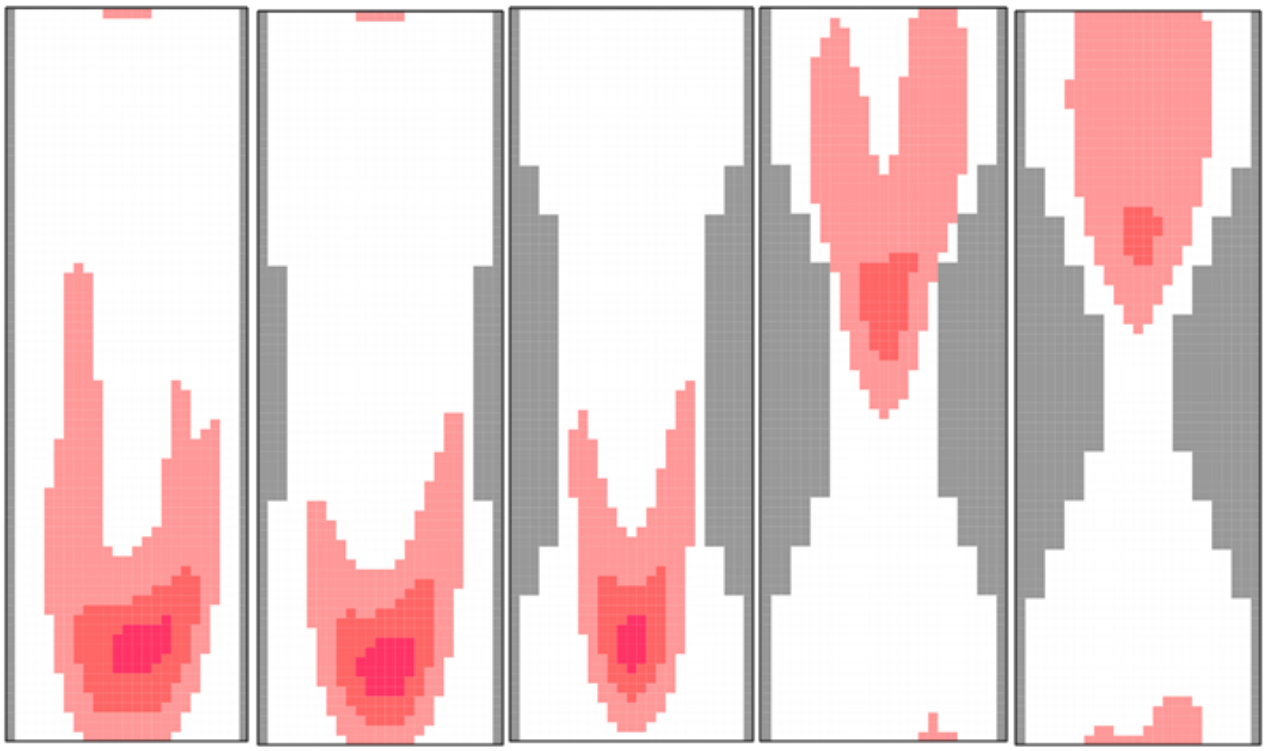

Figure 34: Simulation of CAD after 150 iterations: 0%, 17%, 35%, 53% and 70% blockage respectively

In figure 34 we see that with a second blockage of 35% there is a further decrease in the flow rate, resulting in less dye passing through the blockage. With 53% and 70% blockage would see very little of the bolus making it through the blocked region and the flow rate being further reduced.

As stated before, there exists no analytical solution for fluid flow. The goal of all CFDs is to provide results that accurately portray reality. The goal of this thesis was to create a CFD solver that could be integrated to solve these flows for any system, specifically for use in biological systems. What we simulated here was a part of a larger system. We looked at how the model would behave if the passage width was decreased by a blockage; the real-world equivalent being CAD and its effect on blood flow through the arteries of the heart. The results generated

matched what we expected to see and even helped provide a visual explanation of the dangers of CAD. As we increased the size of the blockage we saw progressively larger decreases in blood flow rate and in the amount of dye that passed through the blockage. However, it was never to the point that the flow was completely cut off and there was still a significant amount of flow passed through the blockages. This is evident in by the boluses in the more severe cases falling significantly farther behind. Furthermore, the shapes of the boluses themselves show that the pressure front, which should be present in all arteries that signify individual heartbeats, have disappeared in the most severe cases. In the real-world this would signify that the oxygen supply to the heart muscles is being compromised. While it is still being delivered to the muscles in the heart an increase in blood pressure would be needed to offset the oxygen shortage due to the blockage. This would add even further stress to the circulatory system. This decreased flow would still likely be enough to maintain regular heartbeat during restful activities, but what happens when the heart rate increases? The muscles will require more oxygen. However, as we see with the 53% and 70% blockages any increase in pressure upstream of the blockage would not result in a change in the rate of flow downstream of the blockage. This would result in the heart muscles becoming ischemic and cause a cardiac event to occur, such as a heart attack. We know that people suffering from CAD show no real symptoms until a cardiac event occurs, which would most likely occur after a period of high exertion.

6. Discussion

The model presented in this thesis met many of the goals we wished to achieve. It was able to provide accurate results in a flexible and efficient manner. This means it would be viable for implementation with larger systems since it would not be too bulky. Also, as previously discussed, the rule-based nature of the model code is significantly is easier to understand. This means users will be able to easily adjust the parameters of the rules to modify the model so it best fits their needs. The model included a function for defining the behavior of fluid barrier-interactions which will ensure it can be used for a wide range of application. Finally, the method of simulation allowed for only the results at the end of each iteration to be stored, and only the density values stored, or whichever values are of interest. This resulted in an average logfile being 1.5 MB which will make it easier to implement with 3D visualization software. The size of the logfile is a relevant benefit. In the work presented in [17] the logfiles generated were too large and required additional post-processing before being integrated with visualization software.

As previously mentioned, by restricting the velocity vector magnitudes it gave rise to the idea of varying time steps. What this means is that during periods of velocities with large magnitudes the equivalent time an iteration represents is decreased. During periods of low flow, the length of time an iteration represents is increased. This variance of time can be handled by the DEVS simulator, which is an important feature, and can help reduce the computational load when it is implemented into a biological system that has periods of high and low flow, a beating heart for example.

The results presented in section 5.2 give a small glimpse into the possible application of the CFD model presented in this thesis. As previously described, the Parametric Human Project [27] is currently developing a database of 3D models of the human body and individual parts. In

the work presented in [17], they demonstrate how Cell-DEVS models can be integrated with other specialized programs to produce realistic and relevant results. In this example, they integrated the model with an architectural program for retrieving the building design and a 3D visualization software (3DS Max) for presenting the results in a more meaningful fashion. The model presented in this thesis is just one more building block into that same framework. The idea behind the DEVS hierarchal architecture is that models can exist as stand-alone models, atomic models, or coupled to be part of a larger whole. This philosophy promotes the idea of using and re-using/recycling previously defined models for new applications. Often the behaviors defined within the model transcend the original goal and can be applied for a wide range of uses. What this means, is that if we wanted to create a model of a beating heart we could do the following: The user loads the 3D model of the heart into the simulation, using the methods presented in [17], couples several existing atomic or coupled models that describe and resolve individual behaviors: such as deformable boundaries, converting boundary deformations to forces, heart valves and CFD. The models would require slight adjustments for the specific parameters desired, however with the rule-based nature of Cell-DEVS models these adjustments are easily made. Finally, the larger more complex model is ready for simulation. The results would then be loaded into the visualization software where the final results are displayed, again possibly implementing the methodologies described in [17]. Overall such a project is large and complex; however, by taking the same approach as the DEVS formalism does to resolving complex systems, by breaking the problem into smaller more manageable parts, it is a realizable goal.

While the scenario presented in section 5.2 does not seem complicated on the outside, a closer examination of its goal emphasizes the merit of such efforts. If the goal had been to simulate the effect narrowing arteries had on Coronary Artery disease, then it may not have been

all together that impressive. Instead, the goal was to create an easy to use CFD model, which could then be used to model biological systems. The solution to fluid flows must be general enough that they could be applied to a range of applications and fields, and that is what was done here. To create the simulation of CFD only the boundary had to be adjusted. The density was easily adapted to represent the tracer dye and the velocity fields used to represent the blood flow. The same model was used previously to simulate a wind tunnel. This was the goal of this thesis, to make an easy to adapt model.

While there were many parts to the model were met or exceeded the goals of the thesis, there were also some opportunities for improvement. The *no-slip boundary* theory was used because it was basic in its nature and therefore easy to adapt for all situations. While for most scenarios it worked well, there were others that it did not function perfectly. For example, with the CAD simulation, when the channel width started to decrease the ratio of boundaries to open space increased. With the increase in boundary cells, it seemed that a small amount of mass was lost due to the boundary. Corners were most likely the greatest culprit since it was theorized that the averaging of the surrounding cells might not offset the density lost to the boundary completely. Never once did this slight loss of mass effect the overall results though. This problem would be worth further research; at a minimum a test of defining other boundary behaviors for the model, if not further implementing different versions of the model, each with different boundary behaviors.

Finally, the execution times of the model need to improve. While the specific times depend on many factors: size, simulation length, and amount of activity. No matter how basic the simulation, the execution length is still too long. For most of the simulations presented in this thesis the execution length was in the range of 40 to 60 minutes. For simulations that lasted 175

frames, this amount of time is too great. As previously mentioned, the method presented in this thesis for implementing the Navier-Stokes equations, was chosen because it should provide the most realistic results in an efficient manner. The natures of the algorithms implemented in this thesis are capable of delivering real-time or near to real-time results. However, in this thesis we were unable to achieve that. It could be that the rules need to be adjusted to be more efficient; however, it is more likely that a new and more efficient method of simulating the results must be created, before any real-time applications of this model can be possible. Once again, the benefits of the DEVS formalism are such that, if a new simulator is developed, by keeping the model and the simulator separate, the model will not require any adjustments to run on this theoretical simulator.

7. Conclusion

There is a need in DEVS-based biological systems modeling, for a method of solving dynamic fluid flows. Such a CFD solver must be able to provide realistic approximations for the flow with minimal computational effort. Furthermore, the model must work for a wide range of applications.

In this thesis, we presented a series of algorithms, based on the Navier-Stokes equations, which approximated the behavior of fluid flows. These algorithms were implemented as a Cell-DEVS using the CD++ Toolkit. The individual algorithms were tested to verify their accuracy. Finally, a simulation was run to examine the effects the narrowing of the coronary artery had on Coronary Artery Disease (CAD). While the model presented here does not contribute specifically to CAD or any biological system in particular, it does contribute a method of solving fluid flows, which can be applied to a wide range of biological systems.

The model presented here is the first approach at using the DEVS formalism to implement a CFD solver. It was successful, in that we were able to implement the algorithms successfully as a Cell-DEVS. Furthermore, the results the model generated were an accurate approximation of real-world flows. Finally, the model was used to demonstrate that it can be used to approximate fluid flows for biological systems.

While the model presented in this thesis represents a good start at developing a method to solve for the fluid flows in large biological systems, it could still be improved in several areas: the need for a 3D CFD solver, deformable boundaries, and adaptation for moving solid objects.

Providing a 3D solver would further widen the possible applications of this model. The nature of the algorithms used in the model presented allow for it to be adapted as a 3D solver. The first and most obvious step would be to add a third velocity component port z to handle the

third directional component. The adding of the third dimension would require that the neighborhood be changed from a 3x3 to a 3x3x3 space. The first change to the algorithm would be to the diffusion function. Two additional cells must be added to allow for diffusion up and down and would be reflected in the following equation:

$$x(i,j,k) = \frac{[x(i,jk)' + \mathbf{a} * (x(i-1,j,k) + x(i+1,j,k) + x(i,j+1,\mathrm{k}) + x(i,j-1,\mathrm{k}) + \mathrm{x(i,j,k-1)} + x(i,j,k+1))]}{1+6a}$$

Equation 18: Adjusted diffusion equation for 3 dimensions

The next function to need changing would be the *advection* function. Instead of the four current cases it takes to define the possible movements, with three dimensions it would require eight. Four cases similar to those presented in figure 16 for when *z* is greater than or equal to zero and less than or equal to one and a slightly varied version of figure 18 for when *z* is less than zero and greater than or equal to negative one as demonstrated in figure 35.

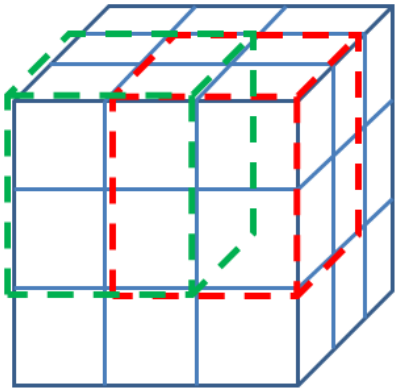

Figure 35: Two of the eight possible cases for the *advection* function

The last two functions to be adjusted are the *projection* and *boundary* functions. For the *projection* function the calculations for *p* and *div* would have to be changed to reflect the addition of the third dimension. The boundary function would be the hardest to adjust for since the loss generated by the averaging of the adjacent cells would become worse since the number

of cells adjacent to a boundary edge and corner would increase. Different boundary behaviors would need to be explored to improve on this.

The implementation of deformable barriers with the CFD would improve the range of applications of the CFD model. This could be done by creating a separate atomic model that is coupled with the CFD to improve its versatility. Its responsibility would be to model the behavior of the boundaries and communicate the changes in shape to the CFD model. More importantly the model would need to solve for the forces generated by the changing of boundary shape and communicate these forces to the CFD model so that the velocity fields can be adjusted accordingly. With this, it would be possible to create more accurate models of systems where the changing of the structure affects/drives the systems.

A final possible extension to the current CFD model would be to model the flow of solid particles. Currently it treats particles as a density of particles, where each cell can hold a multitude of particles. It would be useful to model larger particles, particles the size of a single cell, and how they are affected by the fluid flow. All that would be required is an adjustment to the density solver. The diffusion step would not be required and the *advection* function could be adjusted so that, instead of taking a weighted average of the source cells to calculate the new density in the destination cell, the cell that would have received the largest portion of the "density" from the particles current location would take the whole particle instead of just a part of it.

References

[1] J.D. Anderson, *Basic philosophy of CFD: Computational Fluid Dynamics, 3rd Ed*., Springer-Verlag Berlin Heidelberg, 2009, pp. 3-14.

[2] B.P. Zeigler, H. Praehofer, T.G. Kim, *Theory of Modeling and Simulation, 2nd. Ed*, Academic Press, London, 2000

[3] G.A. Wainer, *Discrete-Event Modeling and Simulation: A Practitioner's Approach*, CRC Press, Boca Raton, FL, 2009.

[4] M. Bonaventura, G. Wainer, R. Castro. "A Graphical Modeling and Simulation Environment for DEVS". *SIMULATION: Transactions of the Society for Modeling and Simulation International.* January 2013 vol. 89 no. 1. pp. 4-27.

[5] J. Stam, "Real-Time Fluid Dynamics for Games" *Proceedings of the Game Developer Conference*, March 2003, San Jose CA, USA.

[6] "Cellular Automata", *Stanford Encyclopedia of Philosophy* [Online] last accessed 2013-12-09, Available: http://plato.stanford.edu/entries/cellular-automata/

[7] B. Chopard, A. Dupuis, A. Masselot, and P. Luthi, **"**Cellular Automata and Lattice Boltzmann Techniques: an Approach to Model and Simulate Complex Systems", *Advances in Complex Systems* 2002 Vol 5, Issue 2, pp. 103-246

[8] D. D'humières, P. Lallemand, "Lattice Gas Automata for Fluid Mechanics", *Physica A: Statistical Mechanics and its Applications*, Volume 140, Issues 1–2, December 1986, pp. 326–335

[9] S. Choia, C. Lin, “A Simple Finite-Volume Formulation of the Lattice Boltzmann Method for Laminar and Turbulent Flows”, *Numerical Heat Transfer, Part B: Fundamentals: An International Journal of Computation and Methodology*, Volume 58, Issue 4, 2010, pp. 242-261

[10] P. Moin, K. Mahesh, “DIRECT NUMERICAL SIMULATION: A Tool in Turbulence Research”, *Annual Review of Fluid Mechanics*, 1998, Vol. 30, pp. 539-578

[11] S. Pope, *Turbulent Flows*, Cambridge University Press, 2000

[12] H. Pitsch,. "Large-Eddy Simulation of Turbulent Combustion". *Annual Review of Fluid Mechanics* vol 38, pp. 453–482.

[13] R. Stoll, F. Porté-Agel, "Large-Eddy Simulation of the Stable Atmospheric Boundary Layer using Dynamic Models with Different Averaging Schemes", *Boundary-Layer Meteorology,* vol 126, pp. 1–28.

[14] M. Van Schyndel, G. Wainer, M. Moallemi, “Computational Fluid Dynamic Solver based on Cellular Discrete-Event Simulation”, Proceedings *of Simultech 2013*, Rejkyavik, Iceland

[15] G. Wainer, “TUTORIAL: Advanced Spatial Systems with Cellular Discrete-Event Modeling and Simulation”, *Proceedings of the 2012 Winter Simulation Conference (WSC),* 2012, Berlin, Germany, pp. 1-15

[16] J. Ameghino, A. Troccoli, G. Wainer, “Models of Complex Physical Systems Using Cell-DEVS”, *Proceedings from the 34th Annual Simulation Symposium*, 2001. April 2001, Seattle, WA,

[17] S. Wang, M. Van Schyndel, G. Wainer, V. S. Rajus, R. Woodbury, “DEVS-based Building Information Modeling and simulation for emergency evacuation”, *Proceedings of the 2012 Winter Simulation Conference (WSC),* 2012, Berlin, Germany, pp. 1-12

[18] G. Wainer et. al, "Advanced DEVS models with application to biology and medicine", *AI, Simulation and Planning in High Autonomy Systems (AIS 2007)*, 2007, Buenos Aires, Argentina

[19] R. Goldstein, G. Wainer, "Modeling Tumor-Immune Systems with Cell-DEVS", *Proceedings of the European Conference on Modeling and Simulation (ECMS),* June 2008, Nicosia, Cyprus

[20] R. Goldstein, G. Wainer, "Simulation of a Presynaptic Nerve Terminal with a Tethered Particle System Model", *31st Annual International IEEE EMBS Conference, 2009*, Minneapolis, MN

[21] A. M. Uhrmacher et. al., "Combining micro and macro-modeling in DEVS for computational biology*", Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, 2007, NJ, USA. Pp. 871-880.

[22] M.A. Day, "The No-Slip Condition of Fluid Dynamics"*, Erkenntnis*, vol. 33, 1990, pp. 285-296.

[23] W. Huang, C. Wu, B. Xioa, W. Xia, *Computational Fluid Dynamic Approach for Biological System Modeling*, Cornell University Library, August 2005

[24] T. Scott, D. Banks, A. Mishra, "Introduction to Applied Computational Fluid Dynamics for the Biological Safety Environment", *Applied Biosafety, Journal of the American Biological Safety Association*, Volume 11-4, 2006, pp.188-196

[25] P. Xu, N. Fisher, S. Miller, "Using Computational Fluid Dynamics Modeling to Evaluate the Design of Hospital Ultraviolet Germicidal Irradiation Systems for Inactivating Airborne Mycobacteria", *Photochemistry and Photobiology*, vol 89- 4, July/August 2013, pp.792–798,

[26] D. Capone, R. Kunz, Computational Mechanics - Biological/Biomedical Flows, [Online], Pennsylvania State University, Available: https://www.arl.psu.edu/fsm_cm_proj_bbf.php

[27] Autodesk Research, Parametric Human Project, [Online] Available: http://phuman.webfactional.com/home/

[28] S. Wang, G. Wainer, V. S. Rajus, P. Woodbury, "Occupancy Analysis using Building Information Modeling and Cell-DEVS Simulation", *Proceedings of 2013 SCS/ACM/IEEE Symposium on Theory of Modeling and Simulation, TMS/DEVS'13, 2013*, San Diego, CA, USA

[29] A. Lopez, *Extending the Cell-DEVS modeling language*, M.S Thesis, Universidad de Buenos Aires, Computer Science Department, 2003

[30] Mount Sinai Hospital, Fighting Coronary Disease, [Online] Available: http://www.mountsinai.org/patient-care/service-areas/heart/areas-of-care/heart-attack-coronary-artery-disease

[31] Mayo Clinic, Coronary Angiogram, [Online] Available: http://www.mayoclinic.com/health/coronary-angiogram/MY00541

A Building Evacuation

A1 Implementation as a CD++, Cell-DEVS Model

The first step to designing a Cell-DEVS model is determine the state-values that will be generated by the local computing function. Each state value will represent a specific situation or behavior that will arise during the simulation. Table 2 outlines the cell states and their corresponding behaviors.

Table 2 Cell States and Definitions

Cell State	Cell Color	State Name	Cell State	Cell Color	State Name
1		Wall	8		Up Occupied
2		Exit	9		Left
3		Down	10		Left Occupied
4		Down Occupied	11		Top of Stairs
5		Right	12		Top Occupied
6		Right Occupied	13		Bottom of Stairs
7		Up	14		Bottom Occupied

All cell based models implemented using the DEVS formalism, or Cell-DEVS, share the same basic template and can be broken into two main parts; the specifications and the implementation of rules.

The specification section of the model is responsible for defining the important information that will be needed for the simulation. For the purpose of this model the dimensions where set to be (109,43,3) and the boarders as *unwrapped*. The definition of the neighbors for this model is somewhat unique. Since we are dealing with moving people and traversing different floors, a unique neighborhood had to be defined. First, to ensure no collisions occur between people, the neighborhood for that plane, or floor, is defined as seen in the right part of Figure 36. Figure 36 shows the current layer neighbors and two additional upper/lower stair

cells, i.e. (0,0,-1) and (0,0,1). These two neighbors allow for people to traverse from one floor to the next.

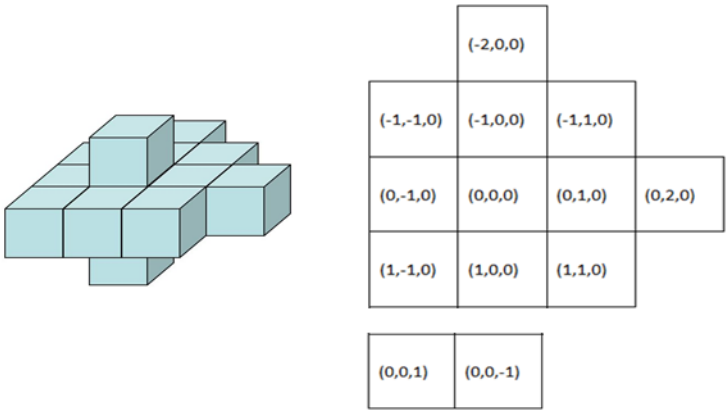

Figure 36: Neighbourhood definition

A typical neighborhood definition would include just the surrounding eight cells, however when movement is involved, often we'll see additional neighbors added. For the purpose of this model the priority or right of way is assigned to certain movement types and are in this order: down, left, up and then right.

A described in section 2.5, the first step in the implementation of the simulation is to overlay a map of the most direct route to an exit or stairwell as well as seed the building with people. The purpose of this is to provide a route for people to follow when leaving the building. This is an iterative process. It starts at an exit or stairwell determining which state behavior; left, right, up or down, would ensure an evacuee would enter the exit or stairwell. This process continues, getting further and further away from the exit, until all cells have been given a state value that will ensure they are able to leave the building. Figure 37 demonstrates how a person (blue) can follow the pathway to exit the building.

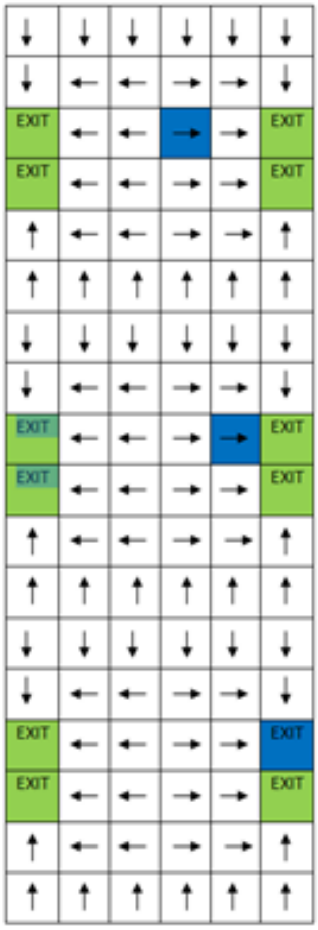

Figure 37: Pathway initialization

The following rules are responsible for initializing the pathway. Besides, the saturation of people in the building is set to 0.005 or 0.5%, however this default value can be adjusted to any desired value.

```
rule : {if(uniform(0,1) < 0.005, 4, 3)} 0 { (0,0,0) = 0 and
(1,0,0) > 1 and (1,0,0) < 10}

rule : {if(uniform(0,1) < 0.005, 6, 5)} 0 { (0,0,0) = 0 and
(0,1,0) > 1 and (0,1,0) < 10}

rule : {if(uniform(0,1) < 0.005, 8, 7)} 0 { (0,0,0) = 0 and (-
1,0,0) > 1 and (-1,0,0) < 10}

rule : {if(uniform(0,1) < 0.005, 10, 9)} 0 { (0,0,0) = 0 and
(0,-1,0) > 1 and (0,-1,0) < 10}
```

The code up to this point can be thought of as the initialization of the model. By now the cell space has been entirely defined and the simulation is ready to begin. The modeling of the evacuees moving through the building is was broken into five events that can occur at a cell; someone entering the cell, someone exiting the cell to an adjacent cell, someone entering an exit, someone entering a stairwell and finally someone exiting a stairwell.

The following are the rules that define the behavior of someone entering a cell.

```
rule : 4  100 { (0,0,0) = 3 and ( (0,1,0) = 10 or (-1,0,0) = 4
or (0,-1,0) = 6 or (-1,0,0) = 14 or (1,0,0) = 14 or (0,1,0) = 14
or (0,-1,0) = 14 )}
rule : 6  100 { (0,0,0) = 5 and ( (1,0,0) = 8 or (-1,0,0) = 4 or
(0,-1,0) = 6 or (-1,0,0) = 14 or (1,0,0) = 14  or (0,1,0) = 14
or (0,-1,0) = 14 )}
rule : 8  100 { (0,0,0) = 7 and ( (1,0,0) = 8 or (0,1,0) = 10 or
(0,-1,0) = 6 or (-1,0,0) = 14 or (1,0,0) = 14  or (0,1,0) = 14
or (0,-1,0) = 14 )}
rule : 10 100 { (0,0,0) = 9 and ( (1,0,0) = 8 or (0,1,0) = 10 or
(-1,0,0) = 4 or (-1,0,0) = 14 or (1,0,0) = 14  or (0,1,0) = 14
or (0,-1,0) = 14 )}
```

In other words: If the cell is currently empty and one of the adjacent cells is full, and the behavior of this cell indicates that they would be entering the current cell; i.e. the cell to the left's state value is 6, then the state value is updated to reflect this, otherwise the cell remains empty.

The following is the implementation of the second scenario, someone leaving a cell.

```
rule : 3  100 { (0,0,0) = 4 and odd((1,0,0)) }
rule : 9  100 { (0,0,0) = 10 and odd((0,-1,0)) and (-1,-1,0) !=
4 }
rule : 7  100 { (0,0,0) = 8 and odd((-1,0,0)) and (-2,0,0) != 4
and (-1,1,0) != 10 }
rule : 5  100 { (0,0,0) = 6 and odd((0,1,0)) and (-1,1,0) != 4
and (0,2,0) != 10 and (1,1,0) != 8 }
```

This is one of the most important sections of the model to not make any mistakes. In the previous section of code, we did not care were the evacuee was arriving from, only that there was someone to enter the cell. It is during this step that we determine which cell is responsible for that person moving. Due to how the rules were defined, in series, there is an inherent priority given to the order of movement. It is also here that the additional neighbors are required to ensure two people do not move into the same cell, i.e. mass is not lost in the system.

The following rules are for someone entering an exit.

```
rule : 3 100 { (0,0,0) = 4 and ((1,0,0) = 2 or (1,0,0) = 11) }
rule : 9 100 { (0,0,0) = 10 and ((0,-1,0) = 2 or (0,-1,0) = 11)
}
rule : 7 100 { (0,0,0) = 8 and ((-1,0,0) = 2 or
(-1,0,0) = 11) }
rule : 5 100 { (0,0,0) = 6 and ((0,1,0) = 2 or (0,1,0) = 11) }
```

The behavior of this section and the following sections is similar to the first. We simply need to track if someone is able to enter the cell from an adjacent cell. The previous set of rules will ensure evacuees do not “disappear” during this process.

The final two sections, entering and exiting a stairwell, are used for traversing floors, therefore unlike evacuees entering an exit, were they just leave the building all together, the evacuees must re-emerge from the stairwells into the floor below. Once someone had entered a cell defined as a stairwell, they wait there until the cell below becomes empty. Once this occurs the evacuee travels to the floor below. This process repeats until they reach the main floor, were they emerge onto that floor and head to an exit. The following rules handle this behavior.

```
rule : 12 100 { (0,0,0) = 11 and ((-1,0,0) = 4 or (0,-1,0) = 6
or (1,0,0) = 8 or (0,1,0) = 10 ) }
```

```
rule : 13 100 { (0,0,0) = 14 and ( (1,0,0) = 3 or (1,0,0) = 5 or
(1,0,0) = 7 or (1,0,0) = 9 or (-1,0,0) = 3 or (-1,0,0) = 5 or (-
1,0,0) = 7 or (-1,0,0) = 9 or (0,1,0) = 3 or (0,1,0) = 5 or
(0,1,0) = 7 or (0,1,0) = 9 or (0,-1,0) = 3 or (0,-1,0) = 5 or
(0,-1,0) = 7 or (0,-1,0) = 9) }
rule : 11 100 { (0,0,0) = 12 and (0,0,-1) = 13 }
rule : 14 100 { (0,0,0) = 13 and (0,0,1) = 12 }
```

From these simple rules a complex evacuation model was created. Furthermore, it was possible to extract real-world information from the results of the model. For example, during the testing of the model, a scenario, as seen in figure 38, was run.

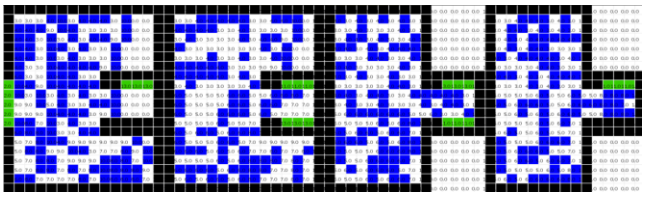

Figure 38: Basic Building design at t=00:00:00:000

A second scenario was executed using the exact same parameters, however with more stairwells. The hypothesis being, that additional stairwells would allow for people to evacuate faster. While this was partially true, the final result was a severe bottleneck near the exit and in the end it took longer for people to evacuate, as seen in figure 39.

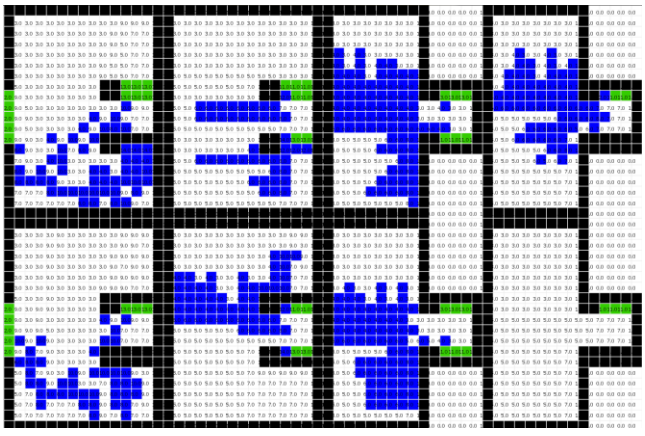

Figure 39: Modified Building (Top) and Original Building (Bottom) at t=00:00:250:000

When the model was run for a real-world building schematic, it was found that the total evacuation time was proportional to the number of occupants in the building, see figure 40.

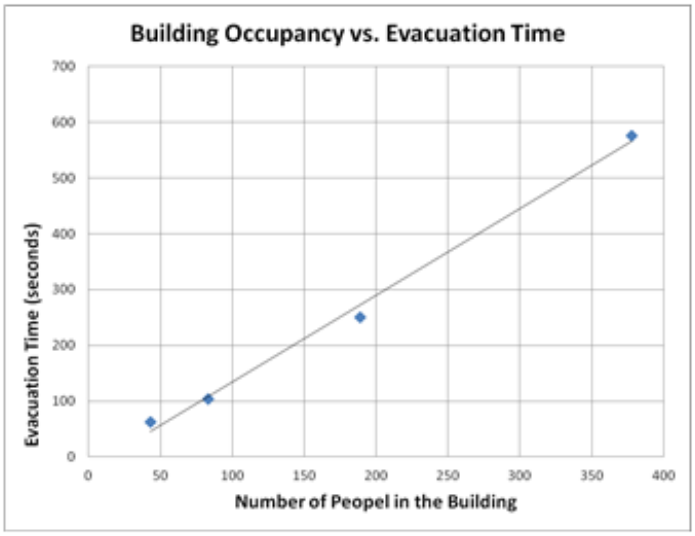

Figure 40: Building Occupancy vs. Evacuation Time

The model created here was proof of the theory behind CA, emergence. From a series of very simple rules, that dictate an individual cell's behavior, a larger more complex set of behaviors can emerge. With these simple rules we were able to generate results that reflect the real-world scenario, as well as provide useful insight, that can be used to develop and test hypotheses.