Advanced Computational Fluid Dynamic Solver Based on Cell-DEVS

Michael Van Schyndel, Gabriel Wainer
Department of Systems & Computer Engineering,
Carleton University, Ottawa, ON, Canada
Center for Visualization, Simulation and Modeling (VSIM)
mvschynd@connect.carleton.ca, gwainer@sce.carleton.ca

Keywords: DEVS, Cell-DEVS, Computational Fluid Dynamics.

Abstract
Computational Fluid Dynamics (CFD) deals with computing the equations of fluid flows using numerical methods instead of partial differential equations. The Discrete-Event System specification (DEVS) theory has already been used to approximate various continuous systems by applying a quantized state system approach. In this research, we experiment with a new method: using Cell-DEVS theory to create a uniform set of rules for CFD to apply to each cell and execute the state changes of the cells asynchronously. This harmonized set of state changes can effectively render the fluid dynamics, by applying the accurate rules that represents the behavior of the fluid.

1. INTRODUCTION

Computational Fluid Dynamics (CFD) solving is the process for calculating and describing the physics of the movement and interaction of fluid flow with the use of numerical methods [1]. Currently there exist no analytical solution; however, various numerical methods have been proposed, including Cellular Automata (CA) [2]. The behavior of the motion is defined with the Navier-Stokes equations, which are a representation of Newton's Second Law of motion. These cells are solved for discrete durations of time and the results rendered to provide results that are more meaningful.

Cell-DEVS is a derivative of the DEVS formalism that implements the CA methodology. The cells are calculated asynchronously, which reduces unnecessary processing burden, with a continuous time base. Each cell is treated as a DEVS atomic model [4] were the state changes are event driven. Cell-DEVS was originally introduced for modeling and simulation of spatial systems however, there has been no research on adopting it for CFD. In n this research, we propose using the Cell-DEVS methodology to implement CFD equations to simulate fluid dynamics. The rule-based nature of cellular model behavior definition provides a platform for spatial behavior definition, leading to easier and faster adoption and implementation of CFD solver algorithms. The asynchronous updating of the cells should help reduce the large computational time required for solving CFDs. Additionally, simulating with continuous time will provide a better resolution then that provided by the CA's discrete time steps.

The CD++ software [3] provides a development environment to create and navigate through the process of Modeling and Simulation (M&S) of a Cell-DEVS model. CD++ is an open-source framework that has been used to model environmental, biological, physical and chemical models as well as many other real-life simulations. The toolkit includes a high-level scripting language keyed to Cell-DEVS, a simulation engine, a testing interface and a basic 2D [5] and 3D [6] graphical interfaces.

The research and solver presented here are an improved derivative of the solver presented in [7]. By using a new solver and implementing a new strategy for defining the rules, it was possible to increase the computational efficiency drastically. We will discuss the framework that can be used to allow the solver to be implemented as an interactive model and how to export the generated data to other graphical environments.

2. BACKGROUND

Fluid dynamic solvers have been used for a wide variety of purposes. The goal is to create a realistic representation of a naturally occurring fluid system such as rising smoke or blowing dust. The flow of fluids can be viewed as solid particles interacting with a velocity field, or as a density of particles. There are different methods for solving the evolution of these fields and densities, such as, the Lattice-Gas method [8], Navier-Stokes Equations [9] and Riemann Solvers [10].

The Navier-Stokes equations were the first to attempt to provide a physical description of fluid motion by applying Newton's second law of motion "with the assumption that the stress in the fluid is the sum of a diffusing viscous term (proportional to the gradient of velocity) and a pressure term [11]. In Sukop et al. [12], the authors presented a method for creating a basic model of 2D fluid flow that maps the possible collisions that can occur. The randomness generated by these collisions is essential to its ability to simulate flows. A similar model was made to represent the effect of polymer chains on fluid flow [13] where a lattice-gas CA was used to provide a 2D model.

In [14], Koelman and Nepveu demonstrated the use of CA to model flow through porous materials. They were able to model a one-phase Darcy automaton based on a Navier-Stokes automaton; however, when they implemented a two-phase Darcy automaton they had to implement simpler local transition rules.

In [15], Stam proposed a new method of resolving the Navier-Stokes equations. A cell lattice is spanned over the simulation window with each cell holding unique information regarding that particular area. Each cell stores a density value and the horizontal and vertical components of velocity (as well as the z component for a 3-dimensional model). The cell spaces are updated simultaneously at discrete time intervals. This algorithm provided realistic results with limited computational effort by utilizing a rather basic set of rules.

In this paper, we are interested in adapting the algorithms presented by Stam [15] and use the models to define a discrete-event CFD solver developed according to the conventions of the DEVS and Cell-DEVS formalism. DEVS formalism [1] has several characteristics which aide in the development of such simulations. The general nature of the specifications of the DEVS formalism, in comparison to other formalisms, allows it to be used in a wide range of simulation methods [2]. Second, the DEVS formalism provides a hierarchal approach for coupling models to create larger, more complex system. Finally, the principle of DEVS is to separate the model code, which contains the real-world system, from the simulator and the time advance functions, which allows models to be coupled that have different time advance functions. This makes the DEVS formalism a powerful tool for simulating large complex systems, such as biological systems.

A Cell-DEVS model is defined as a lattice of cells holding state variables and a computing apparatus, which is in charge of updating the cell states according to a local rule. This is done using the current cell state and those of a finite set of nearby cells (called its neighborhood), as done in Cellular Automata. Cell-DEVS improves execution performance of CA by using a discrete-event approach. It also enhances the cell's timing definition by making it more expressive. Each cell is defined as a DEVS atomic model, and it can be later integrated into a coupled model representing the cell space. Cell-DEVS models are informally defined as shown in Figure 1.

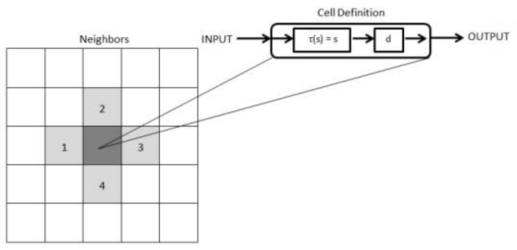

Figure 1. Cell-DEVS Model.

Each cell of the lattice contains information regarding its neighborhood and its local computing function. This local computing function has three 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 the cells to be calculated asynchronously and then updated all at once. This feature allows for the possibility of parallel computing.

The version of the CD++ modeler used to implement the model presented in this paper provides several useful benefits. One such benefit is that the new simulator provides two ways of implementing state variables. The first is a local variable that can only be accessed by the cell to which it belongs. The second is neighbor ports; hereafter referred to as ports, that functions similar to a variable, however, it can be accessed by any cell within the designated neighborhood. These methods improve the efficiency of the model when large amounts of data is required to be stored, since the old simulator required the use of multiple planes to store additional information.

The algorithms by Stam are stable, and allow simulations to be advanced using arbitrary time steps. This feature is relevant to the time advancement strategies facilitated by the DEVS formalism. Second, the relative simplicity of these algorithms lends itself well to the prospect of extending this solver to handle increasingly complex scenarios. Third, these algorithms can be performed using a standard PC for reasonably sized grids of both two- and three-dimensions. Fourth, as the complete C-code implementation of these algorithms is published in [15], it can be used for verification of the Cell-DEVS implementation.

In the following section, we will introduce the definition of this CFD model, and will discuss the model implementation and simulation results.

3. MODEL DEFINITION

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 [15].

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

Equation 1. Velocity Equation in compact vector notation

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

Equation 2. Density Equation in compact vector notation

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 [15]. 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.

The algorithm was broken into two parts: A *density solver* and a *velocity solver*. Each section represented one of the Navier-Stoker equations. The two sections were further broken into discrete steps that must be completed before moving on, as seen in Figure2.

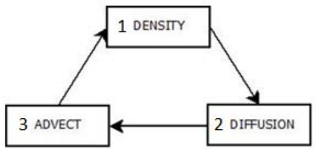

Figure 2. Velocity Solver Steps.

As seen in Figure 3, there are three functions used for both sub-routines: diffusion, advection and projection.

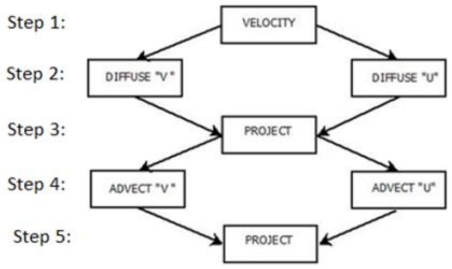

Figure 3. Density Solver Steps.

The diffusion function is responsible for calculating the natural flow of the particles regardless of the forces exerted by the velocity fields. The density for the cell is calculated as the sum of the densities not exiting the cell to the surrounding area and the densities entering the cell from its neighboring cells, as seen in equation 3. Equation 3 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 plus what will enter the cell from the four cardinal neighbors (North, South, East and West). The amount of density that moves between the cells, or viscosity, is determined by the variable a. To increase the resolution of the model this step is run multiple times [15].

$$x(i,j) = \frac{\lfloor x(i,j)' + a * \left(x(i-1,j) + x(i+1,j) + x(i,j+1) + x(i,j-1)\right)\rfloor}{1+4a}$$

Equation 3 Diffusion Calculation [15].

The advection function's role is to apply the forces generated by the velocity fields. The force acting on the density at any location is equal to the equivalent velocity vector of u and v. To apply the forces is significantly more complicated. The simplest approach would be to determine the destination based on the magnitude of the forces applied. However, since the system is treated as a cell space and not all densities will end up in the exact center of the cell after moving, this would cause problems. Instead, to move the density, one simply traces backwards from the cell center to compute where the density would have to come from, as seen in Figure 4 [15].

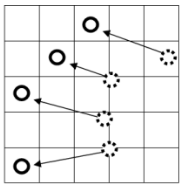

Figure 4. Tracing backwards to the source of density.

Then, we 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 5 shows the movement of densities through a fixed vector field.

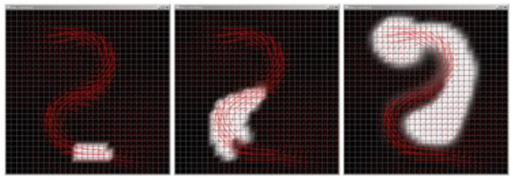

Figure 5. Moving densities, fixed velocity field [15].

During the calculations of the previous steps the results are rarely mass conserving, an important characteristic to maintain realism and stability in the model. The projection function helps conserving mass, and it 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 called twice to help maintain accuracy after the advection step.

Finally, the behavior between the CFD and its boundaries must be defined. For this paper, we chose the *no-slip condition* to model velocity boundary interactions. The theory of the *no-slip condition* states that the fluid velocity is always zero at the boundary-fluid interface [16]. For instance, in Figure 6, at the surface of the interaction between cell **A** (boundary) and cell **B** (a fluid), 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**. In this way, where the two cells are averaged, the result would be zero. The interactions between cells **A** and **B** and between cells **C** and **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 most difficult situation to define 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 passageways to be one cell wide.

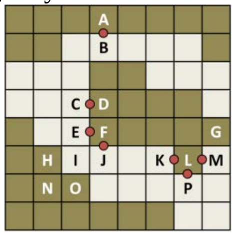

Figure 6. Model of Boundary Fluid Interactions. Boundaries: dark cells; Fluids: light cells.

4. MODEL IMPLEMENTATION

As seen in Figures 2 and 3, the model can be broken into several functions that are used in both sub-routines. Each individual function is used to resolve one of the terms from equations 1 and 2. The specific code for the functions will vary slightly depending on which sub-routine it appears in, however the core of the code will remain the same.

4.1. Diffusion

The diffusion function is responsible for calculating the natural flow of the particles regardless of the forces exerted by the velocity fields. The density for the cell is calculated as the sum of the densities not exiting the cell to the surrounding area and the densities entering the cell from its neighboring cells, as seen in equation 3.

The rate at which the densities leave the cells is referred to as the viscosity and is incorporated into equation 3 as the value for **a**. By incorporating the viscosity into the equation, it is possible to simulate fluids with different parameters. Densities with low viscosities would move through the fluid with very little diffusion; similar to a liquid moving through a rigid space, while densities with high viscosities would rapidly diffuse and take the shape of the container; which is similar to the behavior of a gas.

The implementation of equation 3 with the Cell-DEVS formalism is straightforward. Two ports are used for storing the information regarding the densities, and they are the *value* and *diffusion* port. The *value* port stores the result generated by the diffusion and the advection step, and it is only updated upon the completion of the advection step. Therefore, it can be thought of as the *x'(i,j)* term from equation 3. The other four values for the *x* terms come from the surrounding neighbors' *diffusion* port values. The following code implements this calculation:

```
~diffusion := ((0,0)~value +
  0.05*((0,-1)~diffusion + (0,1)~diffusion
+(-1,0)~diffusion+(1,0)~diffusion))/1.2))
```

For this instance, the viscosity was set to 0.05. Figure 7 below shows the evolution of the densities over time, absent of the presence of any forces generated by the velocity field:

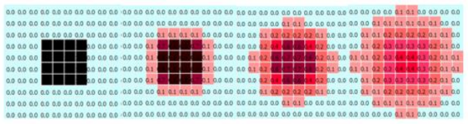

Figure 7. Diffusing densities over 16 iterations with a viscosity of 0.05. Dark to light shades represent a high density to low density gradient.

As we can see, without any external forces, the densities diffuse outwards evenly from a high to low density. It is important to note that the eventually the densities will not appear anymore. This does not mean that there is no density for that cell; it could just mean that the magnitude of the densities is such that it is negligible.

4.2. Advection

The advection step is responsible for the movement of densities and velocity fields. The most obvious method of determining where a *density* will end up is to trace it forward based on the velocity field. However, the method described by Stam [15] suggests that one start in the center of the cell space and trace backwards to find the origins, based on the velocity field. Then, the weighted average of the four closest cells is calculated to determine the source density. This is done because the source is unlikely to be located directly in the middle of the cell, and therefore the surrounding densities will affect the new density values. The advection step, as it appears in the original algorithm in [15] is used to trace the origins of the current density by looking at the vector field. Since the origin is not likely to be at a cell center, a weighted average of the surrounding four cells is taken, their weighting being dependent on their proximity to the origin location.

A logic map is implemented to determine which cells are used based on the state values for the velocity vector component ports, as seen in Figure 8.

In Case #1, the four cells that are circled will be used when $0 \leq u \geq 1$ and $0 \leq v \geq 1$. In Case #2, the four cells that are circled will be used when -1 $<u>$ 0 and 0 $<v>$ 1.In Case #3, the four cells that are circled will be used when $-1 \leq u \geq 0$ and $-1 \leq v \geq 0$. In Case #4, the four cells that are circled will be used when 0 $<u>$ 1 and -1 $<v>$ 0.

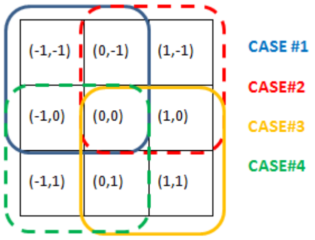

Figure 8. Break down of the possible sources of the density for the advection step.

When implemented, a series of *if* statements is used to determine which case is required and therefore which cells are going to be used in the calculation of the new state value for the *value* port. The implementation is as follows:

```
    if( ( (0,0)~u >= 0 and (0,0)~u <= 1 ),
       //This can be either #1,#4
    if( ( (0,0)~v >= 0 and (0,0)~v <= 1 ),
    //If yes 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+   ...
    )
    )),
       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 + ... )
    ) ,
       // if v is positive than it is Case #2
    (abs((0,0)~u)*(
    abs((0,0)~v)*(0,-1)~diffusion + ...)
    )))
```

The case numbers match those referred to in Figure 8. The calculations are done by directly calling the *u* and *v* vectors. The vector components' magnitudes are equal to *s0* and *t0* respectively, which for the cases where the vector magnitudes could be negative the absolute value of the vector is calculated by calling the *abs()* function. To calculate *s1*and *t1* it is as simple as doing one minus the magnitude of the vector. This method does eliminate some unnecessary calculations since the *(i,j)* values are inherently stored during the simulation process. The values for the densities are grabbed from the *diffusion* port, since it is behaving as the *d0* for this model, and being stored in the *value* port, *d(i,j)*.

4.3. Projection

The projection function is responsible for calculating the first term of the Navier-Stokes equation, as seen in equation 1. This term is responsible for the conservation of mass. The implementation of the projection function is done using two ports, while the remainder of the function is done within the two vector ports. We used two ports to reproduce the original algorithm. The *div* port is responsible for the first part of the computation; hereafter referred to as the div function, while the *p* port handles the second part; hereafter referred to as the p function. The third part is handled by their respective vector component ports.

The implementation of the div function is straightforward. The **h** term is defined as the inverse of the number of cells in the cell-space. This term is later cancelled out when the vectors are separated back to component form. The value for the *div* port is calculated as the negative of the sum of the differences of the horizontal and vertical neighbors, which is scaled by the size of the cell space, **h**. The following is the implementation of the div function:

```
~div  :=  -0.5*(1/441)*(  (1,0)~u  -  (-1,0)~u  +
(0,1)~v - (0,-1)~v)
```

The *p* port value is calculated as the sum of the corresponding *div* port state value and the values of the *p* ports of the immediately adjacent cells, divided by four. This process is run for several iterations before proceeding to the final stage, as follows:

```
   ~p :=((0,0)~div + (-1,0)~p + (1,0)~p +
   (0,-1)~p + (0,1)~p)/4)
```

Once the p loop has been completed, we proceed to the final stage where the results are once again split into component form. The *u* port is updated by taking the current value and subtracting the difference of the horizontal neighbors, which have had the scaling factor *h* removed from them. The *v* port is similarly calculated, using the vertical neighbors instead.

```
   ~u := (0,0)~u - (0.5)*(441)*
   ( (1,0)~p - (-1,0)~p )
   ~v:= (0,0)~v - (0.5)*(441)*
   ( (0,1)~p - (0,-1)~p )
```

During the projection stage, we added the velocity vectors together to make a single velocity field. However, for the rest of the algorithm we want to have the velocities in separate fields. These steps are used twice for each cycle.

4.4. Boundaries

The *no-slip condition* states that the velocity should be average to zero along the boundaries. To implement this we added an addition function to both sub-routines called *boundary* and a port with the same name. The port calculations for the *boundary* variable only occur when initializing the cell-space. It is here that the boundaries and obstacles are created for the model, and they provide a reference location of these obstacles. For example, to create boundaries along the borders of cell-space the following code is used:

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

The *boundary* function is only used in the density solver sub-routine twice. After the diffusion of the densities and the advection of the densities. The purpose here is to ensure that the *value* port remains empty in cells that contain a boundary. This is only a precaution since the velocity fields should stop the densities from entering these cells. Additionally, to ensure mass is not lost to densities being diffused into the boundary cells we, the *diffusion* port is set to equal the average density of its neighboring cells; which with this method of implementing boundaries is never more than two. The following code is how this is implemented:

```
    ((0,1)~diffusion*(  1  ((0,1)~boundary)/2)  +
(0,-1)~diffusion*( 1 - ((0,-1)~boundary)/2) ... )
```

Instead of using a set of *if* statements to covers all the possible set ups, this equation handles all of them The *diffusion* values for the four adjacent cells are summed, with specific cells being zeroed if they are a boundary cell, and then divided by the number of non-boundary cells. Similarly, the *boundary* function zeroes out the cells that are defined as boundaries for the *div* port, again just as a precaution, and ensure there is no mass lost to the system.

The more 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, you 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 way if you were to average the two values the result would be zero. This zeroing of the velocity field will stop the densities from interacting with the boundaries. To make it easier to implement, if we ensure that no boundaries are a single cell thick, we do not have to worry about a situation arising similar to that of L in Figure 6. An example of how this is implemented is as follows:

```
if( (0,0)~boundary = 2,
//If the cell is a boundary cell then,
if( (0,1)~boundary !=2,
//If the cell to the right is empty
-1*(0,1)~v,
if( (0,-1)~boundary != 2,
//If the cell to the left is empty
-1*(0,-1)~v,0))
 (0,0)~v)
//If neither case is true, no changes
```

This process is repeated for the *u* port values as well.

Before the functions can be implemented, we must first initiate the parameters of the model. The first parameter is the neighborhood. As the advection routine sometimes makes use of additional cells, the model uses the nearest neighbors. Secondly, to store all the necessary information and perform the calculations required in each function requires the use of seven ports: *value, diffusion, u, v, p, div* and *boundary*. The local computing function for this model will be defined as a single rule. Since a single frame of the simulation cannot be completed in a single time step, the run during specific periods as outlined in table 1.

Table 1. Timing information for functions during one execution frame lasting 20 time steps.

	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

As you can see after 20 time steps, the routine is completed. Therefore, time steps with a multiple of 20 are regarded as a single frame.

5. RESULTS

In order to test the model, we executed several simulations scenarios, adjusting the velocities, densities and viscous properties. The simulations are uploaded and executed remotely on the RISE server [17] using the new version of the CD++ simulator presented in the introduction. The results are then downloaded and visualized using a 2D tool, as seen in the different figures presented in this section.

The first simulation presented here includes a test case presented in [7], which we used to verify the new model (which does not present differences with the current results. The initial conditions include a single focus of densities and a non-uniform velocity field. The velocities are between the range of $0.6 < u < 0.8$ and $0.1 < v < 0.3$, or $-0.2 < u < -0.4$ and $0.6 < v < 0.8$. Figure 9 shows the results of this simulation (the arrows represent the vector force being applied to the density focus).

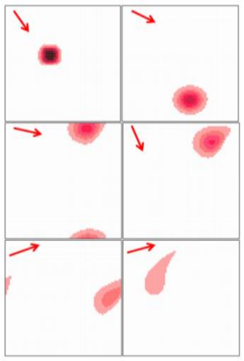

Figure 9. First Test with non-uniform velocity field and viscosity of 0.05.

The new model was able to provide results with a better resolution and a better representation of the evolution of the velocity field. Additionally, the results were generated in a fraction of the time. Originally, we could only run a 50-by-50 cell space (2500 cells), as it was built using seven additional layers, each of them used to use a different piece of information needed by the model (17500 cells). For a reasonable number of frames it would take between 6-8 hours to complete a simulation. The new model is able to complete a simulation of similar size and length in less than one hour.

The next two simulations were to used test the new addition to the model, boundaries. The scenario was set up similar to a wind tunnel were the velocity field was manly wind blowing in 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. For this reason, instead of a constant input of blowing force on the velocity field, similar to a true wind tunnel, a single initiation of the cell space was used. Figures 10 and 11 show the results for these tests with arrows used to represent the forces applied by the velocity fields.

In Figure 10, 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 densities and they are 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.

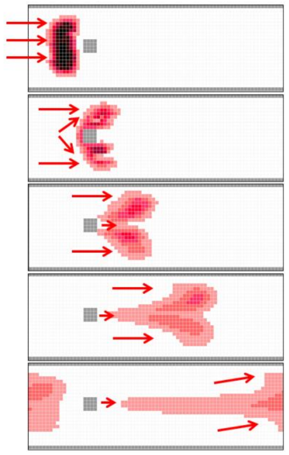
Figure 10. Second Test scenario with small obstacle.

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 on one path.

In Figure 11, 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.

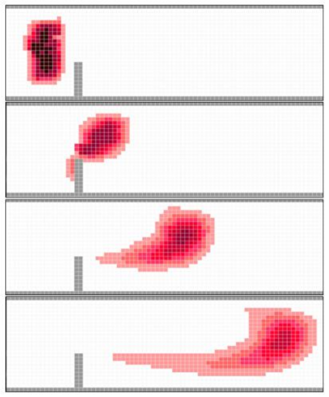
Figure 11. Third Test scenario with large obstacle.

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 described in Figure 11. The other interesting behavior to note is the one of the velocities behind an obstacle perpendicular to the direction of travel. In Figure 10, 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 take 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.

There are three functions that produce the specific results that can be seen in the two simulations. 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. 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 left.

As previously described, the projection function was responsible for ensuring that the fields remain mass conserving and for visual effects. Due to the nature of how the advection function was implemented, if the system were not mass conserving we would see densities disappearing in regions where the velocities are larger than allowed, since there are no conditions to handle this. Clearly, the projection function is performing its most critical role since no velocities are exceeding a magnitude of 1, which considering the initial values ranged from 0.9 to 1.0 would definitely occur if the system were not mass stable.

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.

6. CONCLUSION

Fluid dynamic solvers are used in a wide variety of application ranging from video games and entertainment to modeling of environmental events and biological systems. In this research, a CFD solver is proposed that uses the parameters of a CA in Cell-DEVS. The asynchronous and more efficient computing grid of Cell-DEVS with the continuous time-base allowed for more realistic simulation of fluid dynamics. We showed how CD++ toolkit was used to implement the Cell-DEVS model of the Navier-Stokes equations for CFD. We were able to create a fluid dynamic solver that met the requirements of a Cellular Automata, demonstrating that it is possible to create models of vary complex phenomenon using a relatively simple technique. The model was a significant improvement to the first version, in that it was able to provide results with a better resolution in a significantly shorter time. The model also improved the size of the log files generated which was a major concern of the last model, without sacrificing the ability to access the high level of detail generated during the evolution of both the density and velocity field. The results shown in this paper demonstrate that it is possible for a CFD model to be created and coupled to help resolve the physics of the fluid flow in any system: biological, environmental, etc.

REFERENCES

[1] Anderson, J. D. "Basic philosophy of CFD." Computational Fluid Dynamics, pp. 3-14, 2009.

[2] Ilachinski, Andrew "Cellular Automate: A Discrete Universe" World Scientific Publishing Co. 2001.

[3] Wainer, G. A. Discrete-event modeling and simulation: a practitioner's approach.CRC, 2009.

[4] Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000) Theory of modeling and simulation. 2000.

[5] Kidisyuk, Kiril, and Gabriel A. Wainer. "CD++ Modeler: a graphical toolkit to develop DEVS models." In Proceedings of the 2008 Spring Simulation multi-conference, p. 8.Society for Computer Simulation International, 2008.

[6] Venhola, Wilson, and Gabriel Wainer. "DEVSView: A tool for visualizing CD++ simulation models." SIMULATION SERIES 38, no. 1 (2006): 133.

[7] M. Van Schyndel, G. Wainer, M. Moallemi. Computational Fluid Dynamic Solver based on Cellular Discrete-Event Simulation. In Proceedings of SIMULTECH 2013.Rejkyavik, Iceland. 2013.

[8] Toro, Eleuterio F. "Rienmann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction 3rd Edition" Springer-Verlag Berlin Heidelberg. 2009.

[9] Saleh, Jamal Mohammed.Fluid flow handbook. New York (NY): McGraw-Hill, 2002.

[10] Currie, I. G. Fundamental Mechanics of Fluids, McGraw-Hill, Inc., 1974.ISBN 0-07-014950-X.

[11] J.M. Saleh, Fluid Flow Handbook, McGraw-Hill, New York, 2002.

[12] M.C. Sukop, D.T. Thorne, Jr., Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers, Springer, 2006.

[13] J.M.V.A. Koelman, Cellular-Automata-Based Computer Simulations of Polymer Fluids, Numerical Methods for the Simulation of Multi-Phase and Complex Flow: Lecture Notes in Physics, vol. 398, 1992, 146-153.

[14] J.M.V.A. Koelman, M. Nepveu, Darcy flow in porous media: Cellular Automata Simulations, Numerical Methods for the simulation of multi-phase and complex flow: Lecture notes in Physics, vol. 398, 1992, 136-145.

[15] J. Stam, Real-Time Fluid Dynamics for Games, Proceedings of the Game Developer Conference, San Jose CA,2003.

[16] J.D. Müller, M. Jitsumura, N.H.F. Müller-Kronast, Sensitivity of flow simulations in a cerebral aneurysm, Journal of Biomechanics, vol. 45, 2012, 2539-2548.

[17] Al-Zoubi, K., & Wainer, G. (2010). RISE: Rest-ing heterogeneous simulations interoperability. In Simulation Conference (WSC), Proceedings of the 2010 Winter . IEEE.