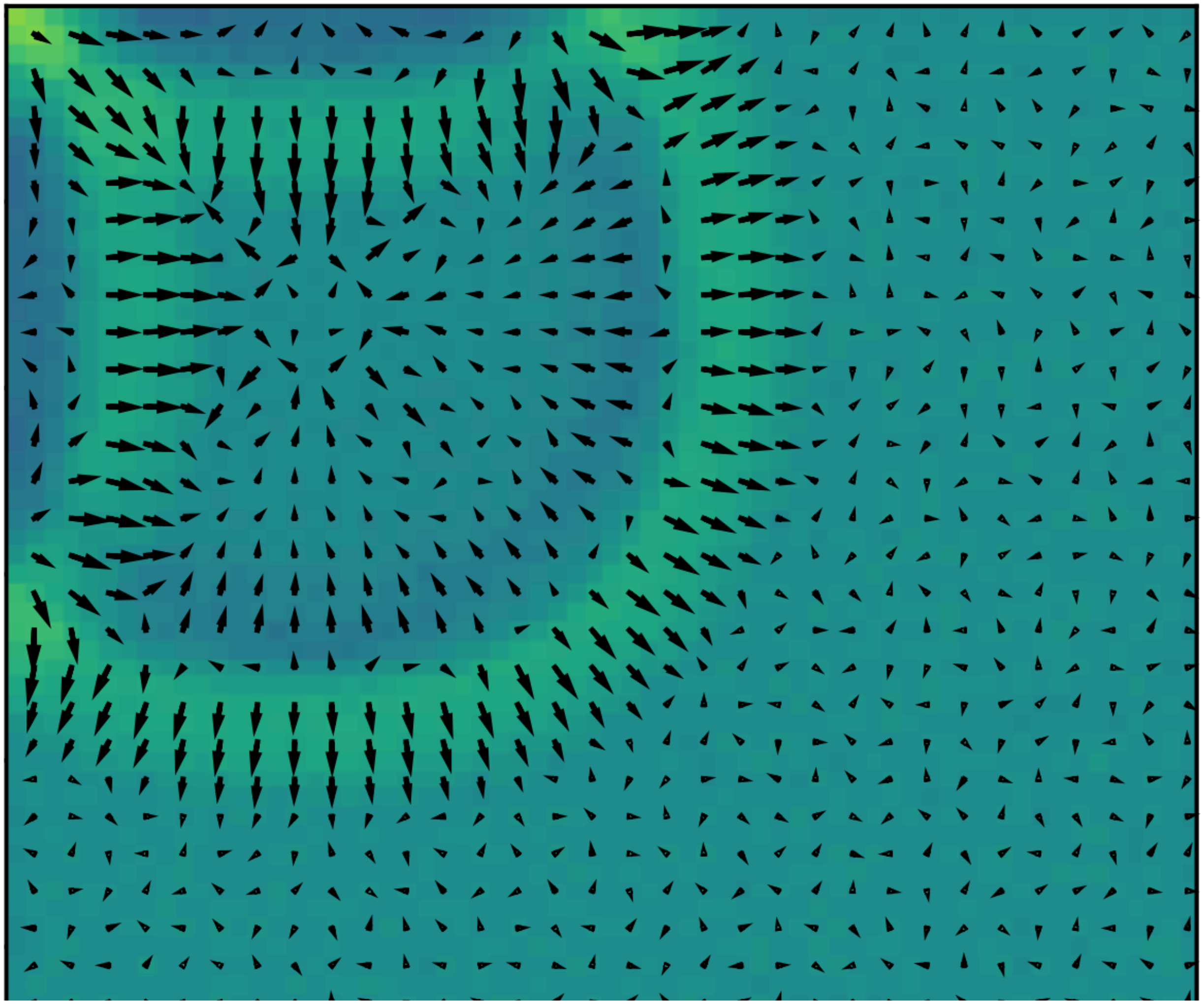
Computer Science Colloquium
Cellular Automata Fluid Dynamics

Dr. Micah Schuster

Summer 2020

Wentworth Institute of Technology



First: Tools

- **An IDE that can handle Java:** I'll use Eclipse since most folks are familiar with it.
- **JavaFX**: For visualization we'll use JavaFX, make sure you can create a new JavaFX project
- **Starter Code (and Finished Code):**
  - **https://github.com/mdschuster/HPPProject**
  - Includes:
    - Starter Eclipse Project
    - Finished Eclipse Project
    - This Presentation

Wentworth INSTITUTE OF TECHNOLOGY

Second: Setup

- **Create Project:** Create a new JavaFX project and copy in the two source files from the github repo.
    - Because of different versions of Java, JavaFX, and OS, you may not just be able to import the Java project directly.
- **Starter Code:** There's a small bit of starter code
    - GridElement class, which mostly represents the squares to draw to the screen.
    - Empty function in main.

Cellular Automata

In CS1 and CS2, I use CA as an example of 1D and 2D arrays, They represent a single simulation in which the states of cells on a board are determined by specific rules.

The best known example of CA is **Conway's Game of Life.**

John Conway developed this cellular automation in 1970. The "game" itself is a zero player game that is entirely determined by the initial configuration of the play area.

Cellular Automata

OXYGEN NOT INCLUDED

Noita

Many games use CA to simulate their physical processes like fluids, gases, granular media, etc.

CA can be very efficient to calculate and games often don’t require physically realistic solutions. It just has to look good enough or act good enough for the simulation.

Cellular Automata

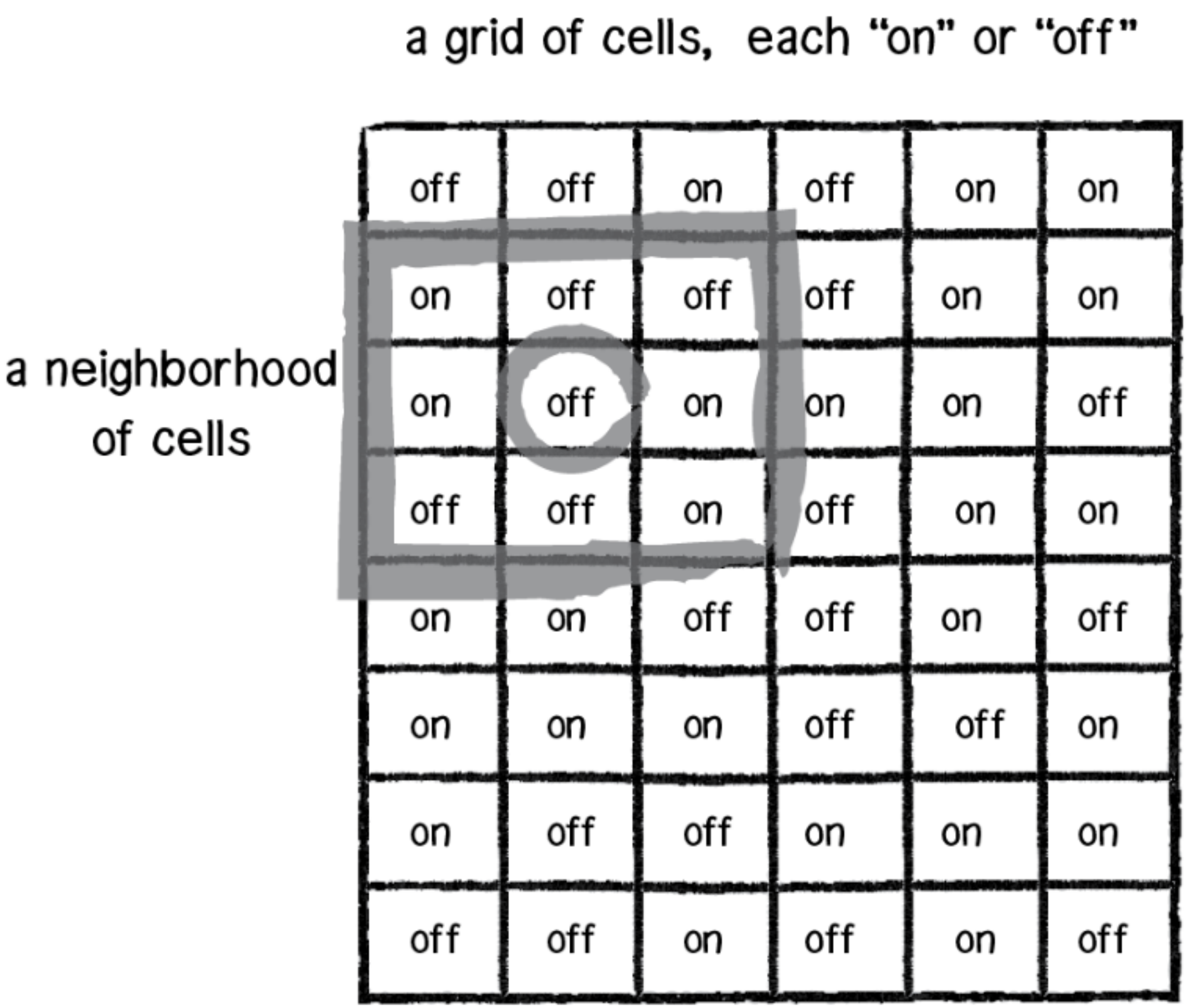
A cellular automation is a model of a system of "cell" objects with the following characteristics:

The cells live on a **grid**
(1D, 2D, 3D, or more)

Each cell has a **state.**
The number of states is usually finite. (e.g. on or off, 0 or 1, alive or dead, etc.)

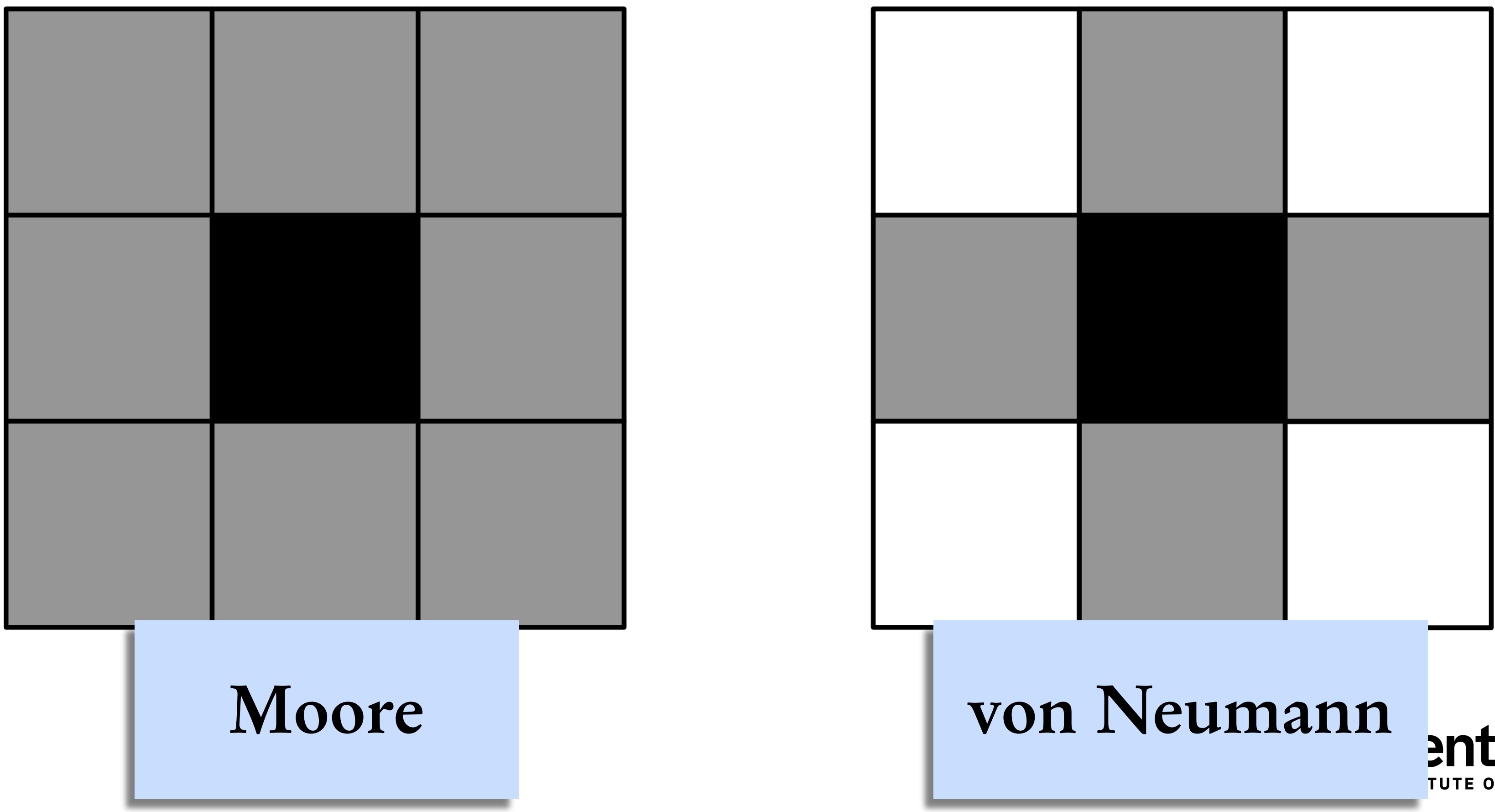
Each cell has a **neighborhood.**
This can be defined in many ways depending on the type of simulation.

Cellular Automata



Cellular Automata

The traditional GOL uses a nine cell Moore neighborhood. However, our fluid model uses a smaller neighborhood, the von Neumann neighborhood:



HPP Model

First CA fluid dynamics simulation

Characteristics:

- Particles move along a lattice
- Each grid **cell** contains 4 **nodes**
- Each **node** is occupied or unoccupied
- Grid updates happen in two steps:
  - Collision - Particles collide within the current cell
  - Propagation - Particles move to new cells

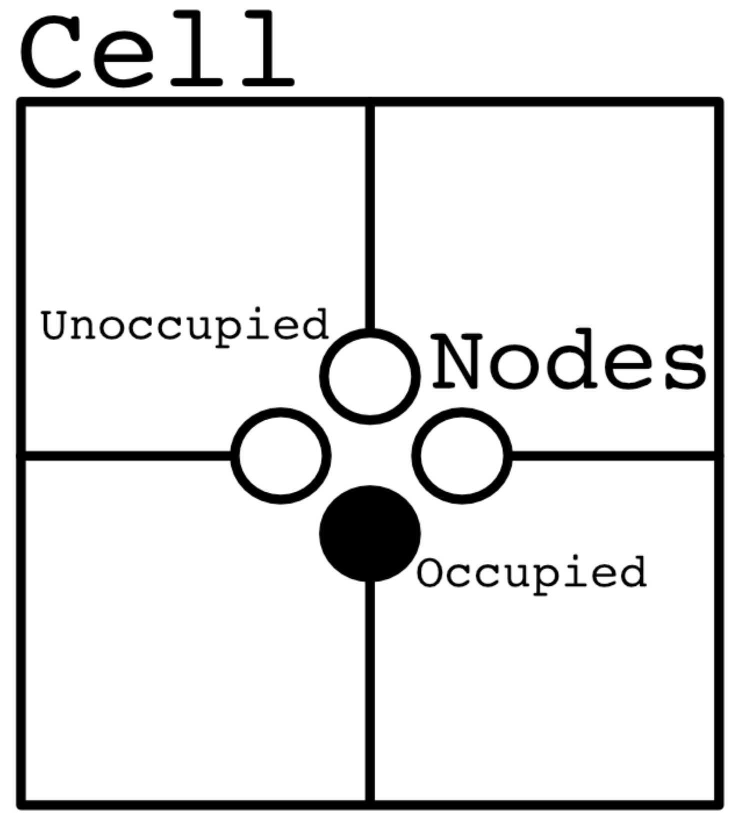
HPP Model

First CA fluid dynamics simulation

The nodes correspond to the cardinal directions

The *node* occupation can be represented by a **single bit**

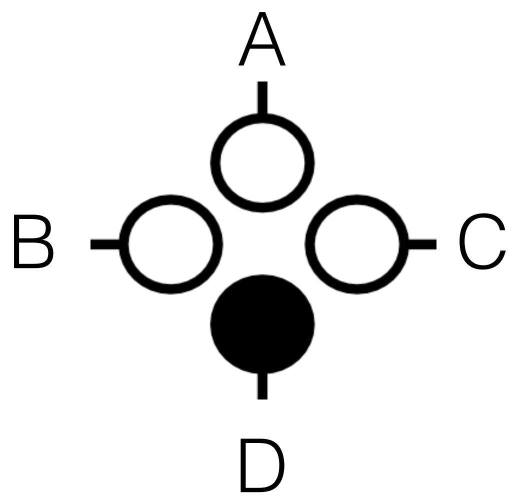
This means the *entire cell* can be stored in a **single byte**



Bitwise Calculations

Operator Name	C/C++ Operator	Binary Example	Hex Example
Bitwise AND	&	`0010 & 0110 = 0010`	`0x02 & 0x06 = 0x02`
Bitwise OR	|	`0010 | 0110 = 0110`	`0x02 & 0x06 = 0x06`
Bitwise XOR	^	`0010 ^ 0110 = 0100`	`0x02 ^ 0x06 = 0x04`
Left Shift	<<	`0011 << 2 = 1100`	`0x02 << 2 = 0x0b`
Right Shift	>>	`1000 >> 3 = 0001`	`0x08 >> 3 = 0x01`
Bitwise NOT	~	`~0110 = 1001`	`~0x06 = 0x09`

We'll use these operators to manipulate the bits within the bytes that represent our cells



1 Byte

`0000 0001`

`XXXX ABCD`

Bitwise Calculations

Operator Name	C/C++ Operator	Binary Example	Hex Example
Bitwise AND		0010 & 0110 = 0010	0x02 & 0x06 = 0x02
Bitwise OR			2 & 0x06 = 0x06
Bitwise XOR			2 ^ 0x06 = 0x04
Left Shift			2 << 2 = 0x0b
Right Shift			8 >> 3 = 0x01
Bitwise NOT			06 = 0x09

Instead of binary, I'm going to use hexadecimal in our code:

$$\overbrace{0101\ 1100}^{\texttt{0x5c}}$$

We'll use thes
to manipula
within the bytes that
represent our cells

D

1 Byte

000 0001

XXXX ABCD

Wentworth
INSTITUTE OF TECHNOLOGY

Grid Setup

```
public void start(State primaryStage){
  GridPane root = new GridPane();
  GridElement[][] grid = new GridElement[SIZE][SIZE]
  for(int i = 0; i<SIZE; i++){
    for(int j = 0; j<SIZE; j++){
      grid[i][j] = new GridElement(i,j);
      root.add(grid[i][j].getGraphic(), i, j);
    }
  }
  for(int i = 0; i<SIZE ;i++){
    for(int j = 0; j<SIZE ;j++){
      byte value = (byte)rand.nextInt(13);
      if(i==0 || j==0 || i==SIZE-1 || j==SIZE-1){
        value = 0;
      }
      data[i][j] = value
      grid[i][j].setOccupation(getBits(value));
    }
  }
}
```

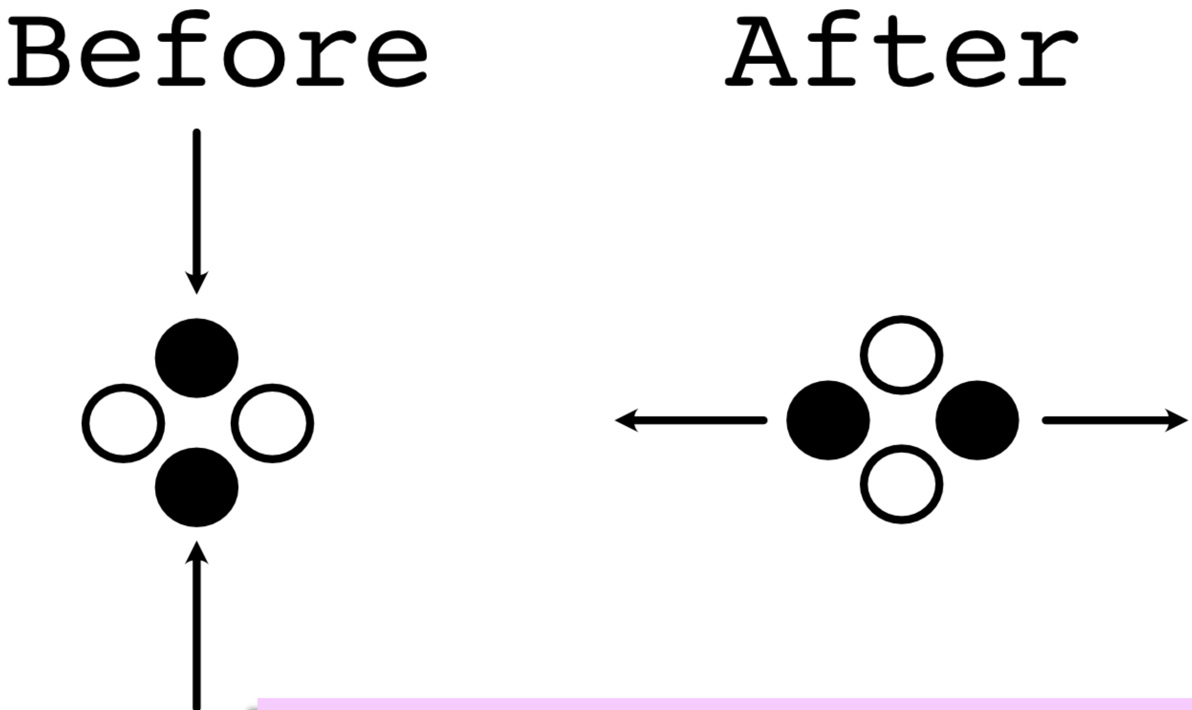
Grid Setup

```
public void start(State primaryStage){
…stuff from previous slide…

  setupCollision();

  //setup high density region
  for(int i=50; i<150; i++){
    for(int j=50; j<150; j++){
      data[i][j]=15;
      grid[i][j].setOccupation(getBits(data[i][j]));
    }
  }

…Handler stuff…
}
```

Wentworth
INSTITUTE OF TECHNOLOGY

Collision

HPP only uses one collision:

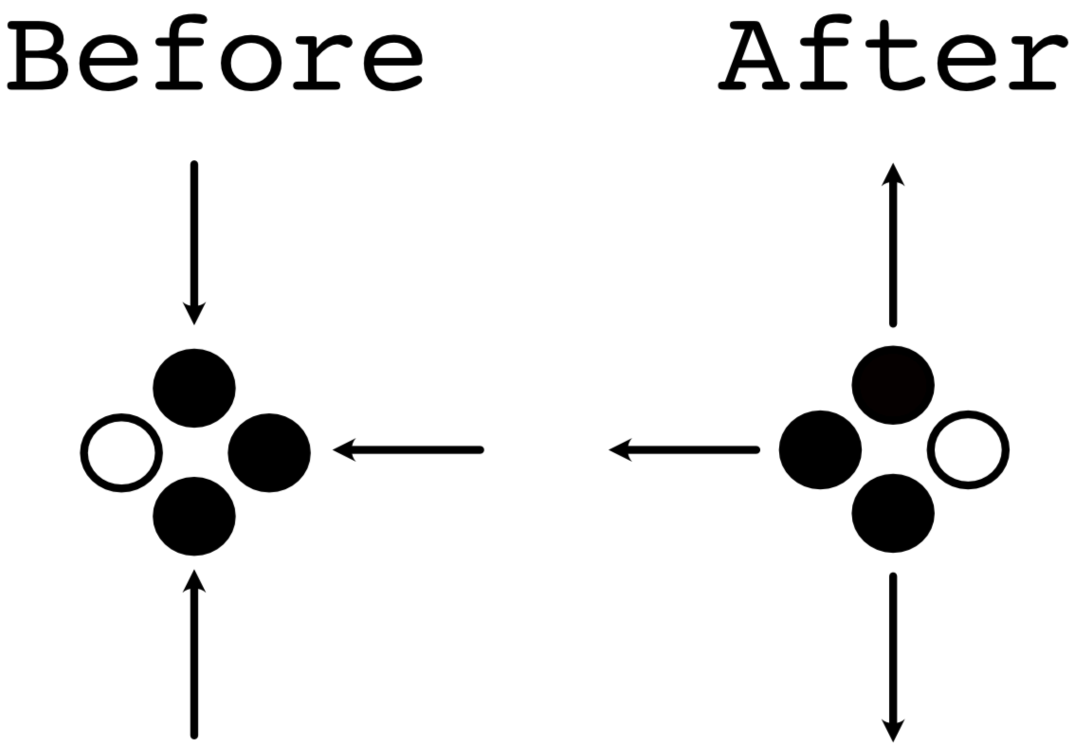


It is rotationally symmetric though!

All other configurations just pass though each other.

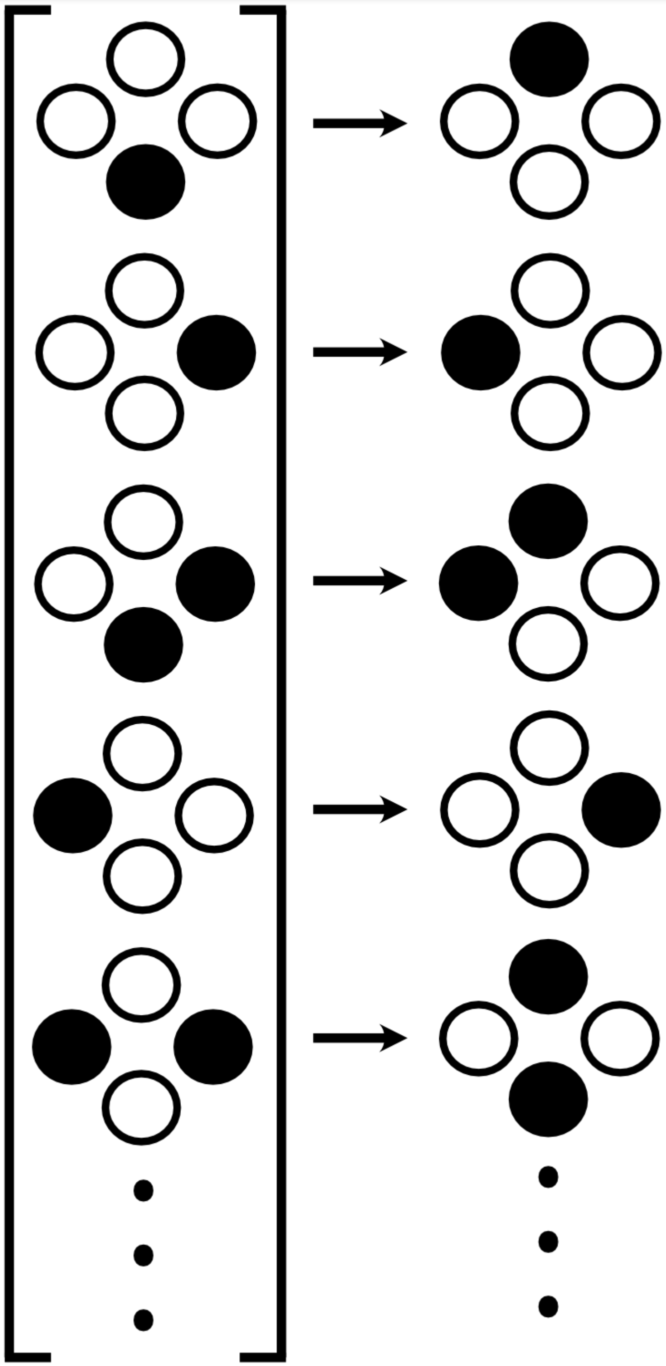
Collision

Pass through example:



While not physically realistic, it does preserve total momentum, which is important for a fluid simulation.

Collision



To represent all the “collisions” we will use a lookup table

This allows us to use the cell value as the index to the lookup table, the value is the new cell value

Collision Lookup

```
public void setupCollision(){
   lookup[0]=(byte)0;
   lookup[1]=(byte)8;   //0001 -> 1000
   lookup[2]=(byte)4;   //0010 -> 0100
   lookup[3]=(byte)12;  //0011 -> 1100
   lookup[4]=(byte)2;   //0100 -> 0010
   lookup[5]=(byte)10;  //0101 -> 1010
   lookup[6]=(byte)9;   //0110 -> 1001
   lookup[7]=(byte)14;  //etc.
   lookup[8]=(byte)1;
   lookup[9]=(byte)6;
   lookup[10]=(byte)5;
   lookup[11]=(byte)13;
   lookup[12]=(byte)3;
   lookup[13]=(byte)11;
   lookup[14]=(byte)7;
   lookup[15]=(byte)15;
}
```

Remember:

```
Node: ABCD
Bin:  0000
Dec:  8421
```

OK, this isn't the best way to write the the function, but it allows me to show the binary for some of the elements

Collision at the Boundary

The first and last row and column will act only as boundaries.

When the "collision" happens, it will bounce back rather than pass through.

Codewise, this just means we don't change the value in that cell

Collision Function

If boundary row/col, keep data the same, otherwise use the lookup.

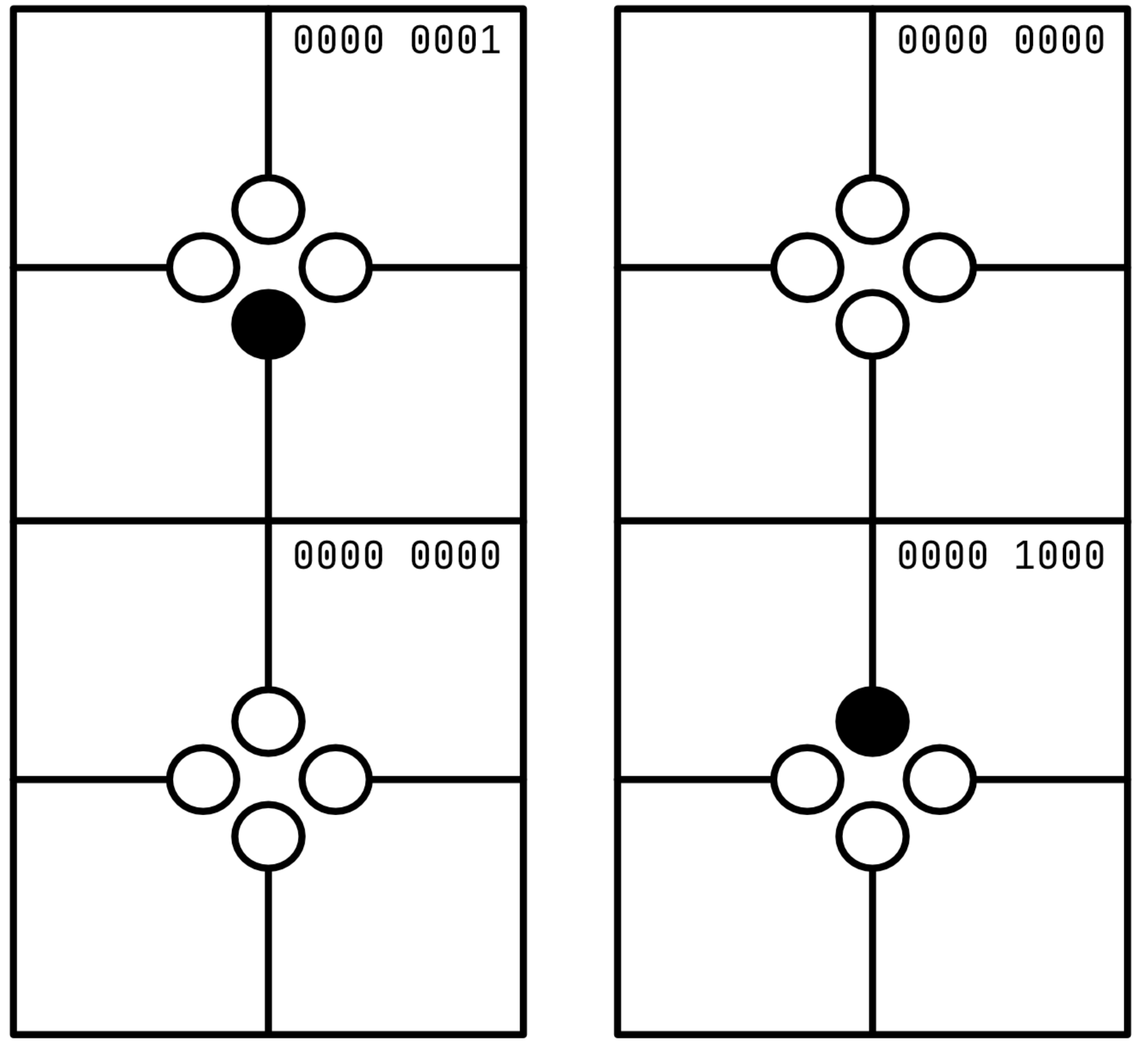
```
public void collide(){
  for(int i=0;i<SIZE;i++){
    for(int j=0;j<SIZE;j++){

      if(i==0 || j==0 || i==SIZE-1 || j==SIZE-1){
        data[i][j]=data[i][j];
      } else {
        data[i][j] = lookup[(int)data[i][j]];
      }

    }
  }
}
```

I don't need the if/else, but it allows the example to be more explicit.

Propagation

Each cell looks to the cells around it (von Neumann neighborhood) to determine its next state.

For example, a **D** bit above your cell will become an **A** bit in your cell.



Before

After

Propagation Function: Masks

```java
public void propagate(GridElement[][] grid){
   //masks
   byte upmask = 0x01;
   byte leftmask = 0x02;
   byte rightmask = 0x04;
   byte downmask = 0x08;

   byte upByte=0, downByte=0, leftByte=0, rightByte=0;

   //more below
}
```

We will “Mask” off the cell to get access to the individual bits that we are looking for

```
Node & Mask = XXXX
1001 & 0001 = 0001
1001 & 0010 = 0000
```

Each mask gets us one of the bits, Then we do a bit shift…

Propagation Function: Masks

```
public void propagate(GridElement[][] grid){
  //masks above…
  for(int i = 0; i<SIZE; i++){
    for(int j = 0; j<SIZE; j++){
      if(i != SIZE - 1)
        upByte = (byte)((data[i+1][j]&upMask)<<3);
      if(i != 0)
        downByte = (byte)((data[i-1][j]&downMask)<<3);
      if(j != SIZE - 1)
        leftByte = (byte)((data[i][j+1]&leftMask)<<3);
      if(j != 0)
        rightByte = (byte)((data[i][j-1]&rightMask)<<3);
      byte value = (byte)(upByte|downByte|leftByte|
                                             rightByte);
      tempdata[i][j] = value
      grid[i][j].setOccupation(getBits(tempdata[i][j]));
    }
  }
  //still more below…
```

Propagation Function: Masks

```
upByte = (byte)((data[i+1][j]&upMask)<<3);
```

What is this actually doing?

Apply the **upMask** (**0x01**) to the cell.

Shift the bits to the left by 3 spaces

```
data[i][j]  =0000 0000
data[i+1][j]=0000 1101
upMask = 0000 0001
```

Cell that we are constructing

Cell above us

```
0000 1101 & 0000 0001 = 0000 0001
0000 0001 << 3 = 0000 1000
```

Propagation Function: Masks

```
upByte = (byte)((data[i+1][j]&upMask)<<3);
```

What is this actually doing?

Apply the **`upMask`** (**`0x01`**) to the cell.

Shift the bits to the left by 3 spaces

Cell that we are constructing

```
data[i][j]  =0000 0000
data[i+1][j]=0000 1101
upMask = 0000 0001
```

Cell

This is a part of our new cell, with a bit in **A**

```
0000 1101 & 0000 0001 = 0000 0001
0000 0001 << 3 = 0000 1000
```

Propagation Function: Masks

```
byte value = (byte)(upByte|downByte|leftByte|rightByte);
```

Now we combine all the individual bits together to make our final cell.

```
upByte = 0000 1000
downByte = 0000 0001
leftByte = 0000 0000
rightByte = 0000 0010
```

Using the bitwise OR allows use to combine these bits together

```
upByte|downByte|leftByte|rightByte = 0000 1011
```

Propagation Function

```
tempdata[i][j] = value
```

Here we use the second 2D array.

We don't want to mess with the **data** array (since we're using it to do our calculations). So our finished values are stored in a temporary array until we're done, then we copy everything back over the **data** array

```
for(int i = 0; i<SIZE; i++){
  for(int j = 0; j<SIZE ;j++){
    data[i][j] = tempdata[i][j];
  }
}
```

Finishing up

```
handler = new EventHandler<ActionEvent>(){
  @Override
  public void handle(ActionEvent event){
    for(int i = 0; i<1; i++){
      collision();
      propagation(grid);
    }
  }
}
```

What does this allow us to do?

The rest of the code should work now. I've already put in a key press for playing and pausing the simulation and the creation of the scene and stage.

Further

You now have the basic HPP model.

There are other things you can add, like obstacles in the simulation domain.

HPP, is not a very good model. It tends to show the underlying grid and doesn't actually solve the Navier-Stokes equations in the continuous limit.

Better models are FHP which is similar to HPP but uses a hex grid with more collisions and Lattice Boltzmann techniques which use a statistical approach rather than individual particles.