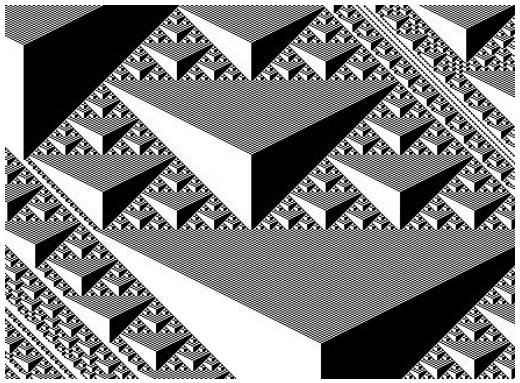
Impact Lab 2025: Programming Fundamentals
Lecture 6: Cellular Automata

Summer 2025

School of Computing and Data Science

Wentworth Institute of Technology



Topics for Today

- **1D Cellular Automata**
- 2D Cellular Automata
- Steering (if we have time)

What is Cellular Automation?

As an example of 1D and 2D arrays, we're going to talk about Cellular Automation (CA). This represents a single simulation in which the states of cells on a board are determined by specific rules.

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.

Noita

OXYGEN NOT INCLUDED

What is Cellular Automation?

As an example of 1D and 2D arrays, we're going to talk about Cellular Automation (CA). This represents 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.

We'll learn the rules and how to create this "game" in **1D** and **2D**!

What is Cellular Automation?

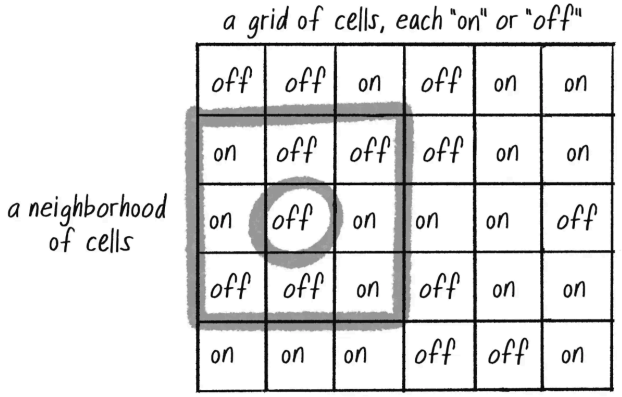
A cellular automation is a model of a system of "cell" objects with the following characteristics:

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.

What is Cellular Automation?



A Bit of History

The development of CA started at Los Alamos National Laboratory in the 40's with Stanislaw Ulam and John von Neumann.

Ulam was interested in crystal growth and von Neumann in self replicating robots.

The most significant work was done by Stephan Wolfram in 2002 with his 1280 page book (*A New Kind of Science*) that explored CA in problems in physics, biology and chemistry.

Elementary CA

What is the simplest CA that you can think of (based on the things we need to make a CA)?

1D Grid
2 States (0 or 1)
3 Cell Neighborhood

Elementary CA

What is the simplest CA that you can think of (based on the things we need to make a CA)?

1D Grid
2 States (0 or 1)
3 Cell Neighborhood

1	0	1	0	1	1	1	0	0	1

Elementary CA

What is the simplest CA that you can think of (based on the things we need to make a CA)?

1D Grid
2 States (0 or 1)
3 Cell Neighborhood

1	0	1	0	1	1	1	0	0	1

Elementary CA

What is the simplest CA that you can think of (based on the things we need to make a CA)?

1D Grid
2 States (0 or 1)
3 Cell Neighborhood

1	0	1	0	1	1	1	0	0	1

What about the edge?

Elementary CA

The most important part of CA - **Time**

1	0	1	0	1	1	1	0	0	1

Generation 1

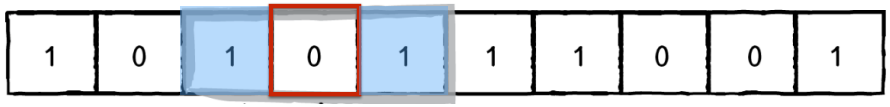
?	?	?	?	?	?	?	?	?	?

Generation 2

?	?	?	?	?	?	?	?	?	?

Elementary CA

We have a description of or neighborhood around a cell, let's use that to determine the next generation!



Notice that our center cell (the 0) determines which cell gets updated in the next generation.

Elementary CA

Thera are many ways to come up with rules that determine the next generation.

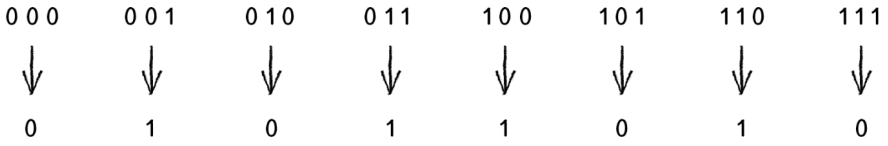
When you blur an image in Photoshop, it uses cellular automata like rules!

For this 1D CA, we can look at all possible configurations of the three cells and come up with rules for each one!

How many rules do we need?

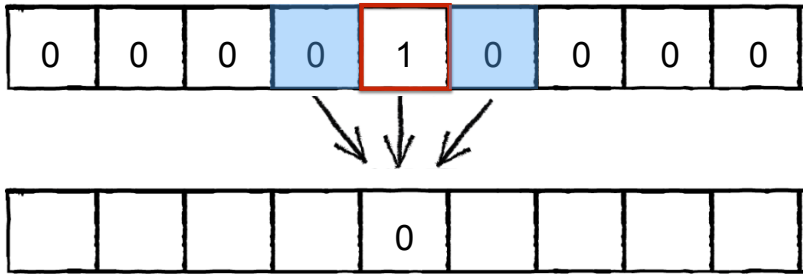
Elementary CA - Rules

8 Configurations:

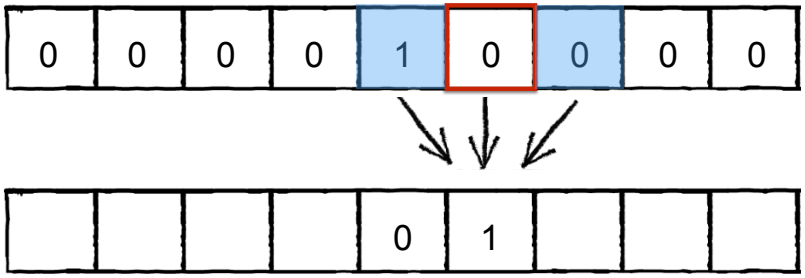


I've picked these rules very specifically...

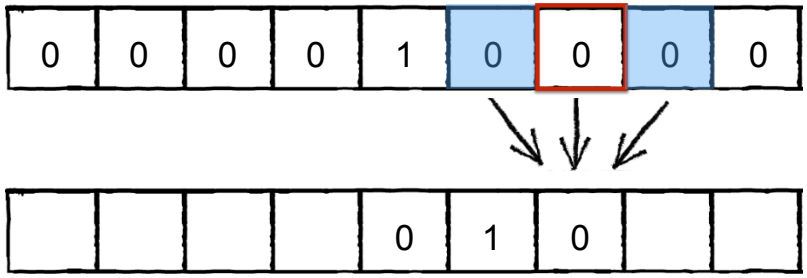
Elementary CA - Handwritten Example



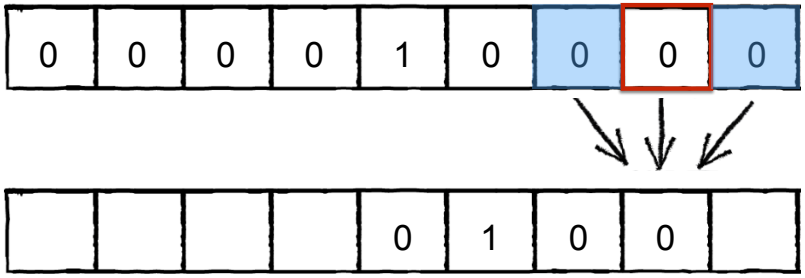
Elementary CA - Handwritten Example



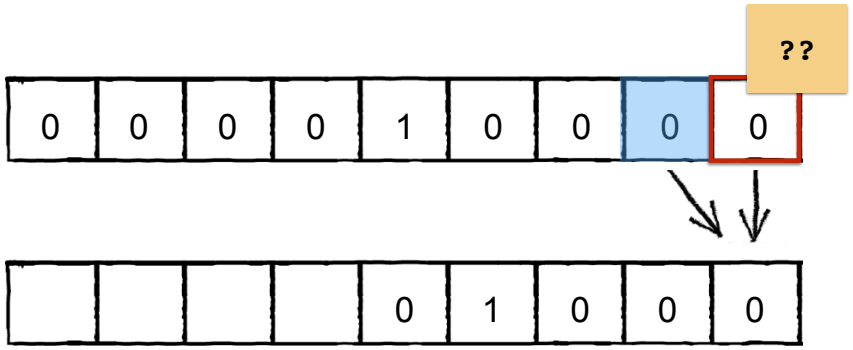
Elementary CA - Handwritten Example



Elementary CA - Handwritten Example

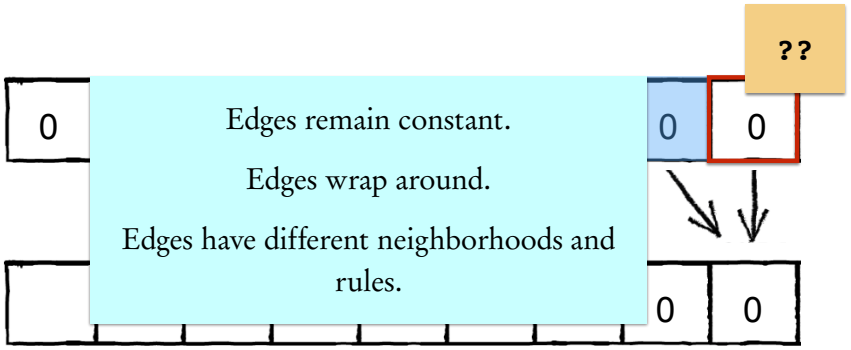


Elementary CA - Handwritten Example



We'll just assume that if we go off the grid, the edge value is zero

Elementary CA - Handwritten Example

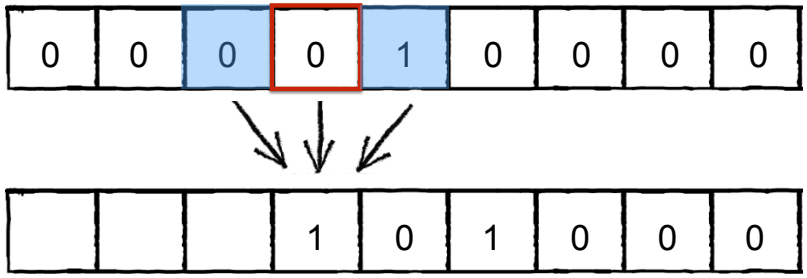


Edges remain constant.

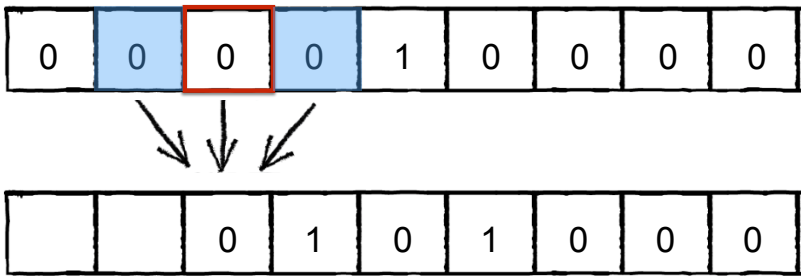
Edges wrap around.

Edges have different neighborhoods and rules.

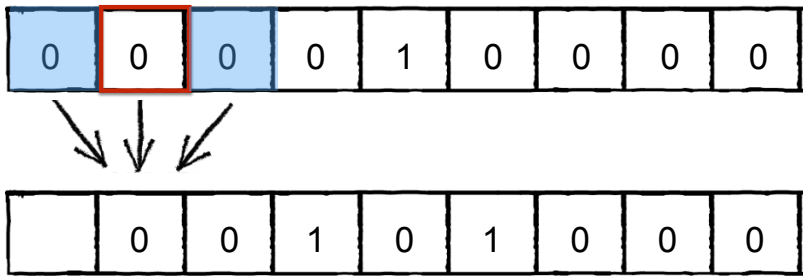
Elementary CA - Handwritten Example



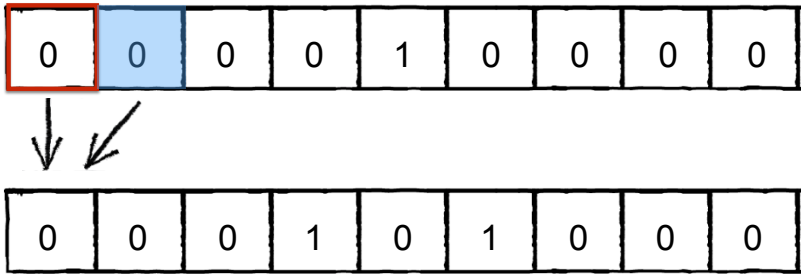
Elementary CA - Handwritten Example



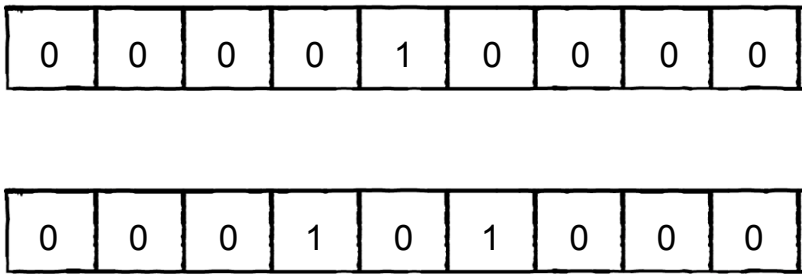
Elementary CA - Handwritten Example



Elementary CA - Handwritten Example



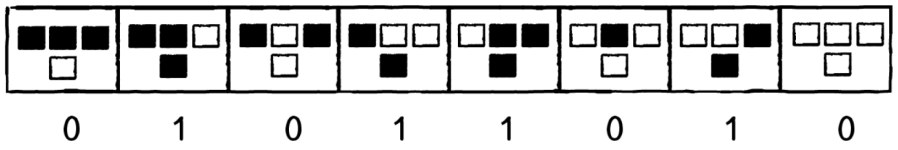
Elementary CA - Handwritten Example



So, we now have a complete first generation. Congratulations, you've done your first CA simulation!

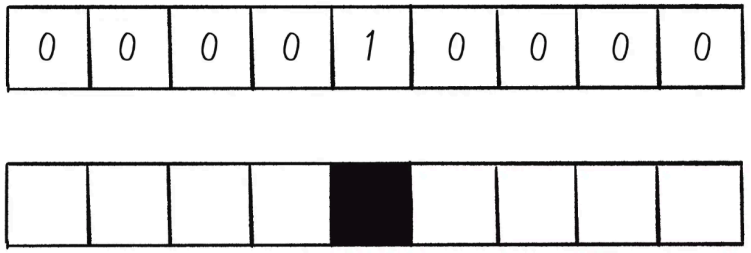
That was a bit tedious though...

Elementary CA - Visual Example



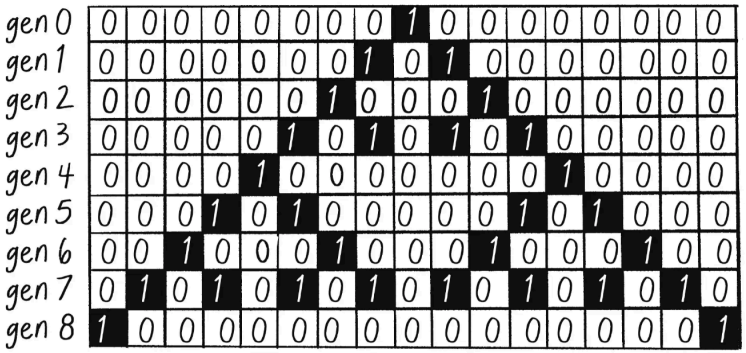
The rules set that we used is called Rule 90. On a computer we can represent 1s as filled cells and 0s as empty cells...

Elementary CA - Visual Example



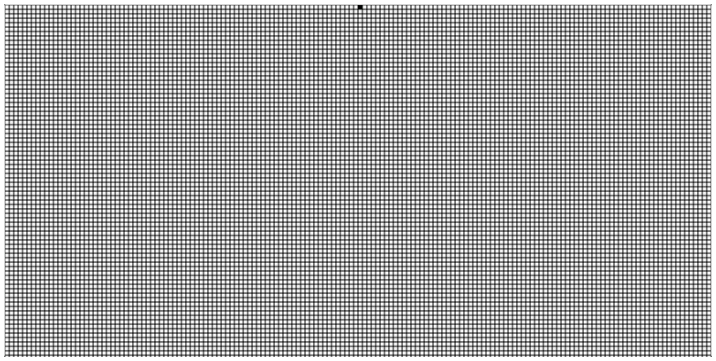
Our initial state would look like this

Elementary CA - Visual Example



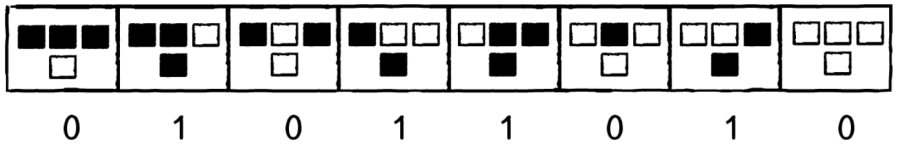
After 8 generations, we get something a bit more interesting...

Elementary CA - Visual Example



We'll code this up soon...

The Ruleset



Rule 90 will be the first one we tackle, but we should allow for any ruleset we want (so we can experiment).

Ideally, we create a function that takes in left, middle, and right and gives is the result.

The Ruleset

To do this, we need to define a ruleset in an easy way:

```
let ruleset=[0, 1, 0, 1, 1, 0, 1, 0];
```

Then, the **`rules()`** function signature can look like:

```
function rules(a, b, c)
```

Within the function, we check what a, b, and c are and return the correct value from the ruleset:

```
if (a === 1 && b === 1 && c === 1) return ruleset[0];
```

Cell Class

- For each cell in the simulation, we can use an object to store the data.
- The constructor defaults the cell state to "dead".
- Show will check the state of the cell, select the correct fill color and and display the rectangle at **x** and **y**.

```
class Cell{
  constructor(x,y){
    this.x=x;
    this.y=y;
    this.state=0;
  }

  show(){
    if(this.state==0) fill(255);
    if(this.state==1) fill(0);
    stroke(0);
    strokeWeight(1);
    rect(this.x,this.y,5,5);
  }
}
```


Cell Class

- For each cell in the simulation, we can use an object to store the data.
- The constructor defaults the cell state to "dead".
- Show will check the state of the cell, select the correct fill color and and display the rectangle at **x** and **y**.

We will create an array of these Cell objects so that we can use them to display the 1D CA.

```
class Cell{
  constructor(x,y){
    this.x=x;
    this.y=y;
    this.state=0;
  }

  show(){
    if(this.state==0) fill(255);
    if(this.state==1) fill(0);
    stroke(0);
    strokeWeight(1);
    rect(this.x,this.y,5,5);
  }
}
```


2D Arrays...

- We can make 1D arrays with a set of square brackets:

```
let grid=[];
```

- To create a 2D array, that we need to visualize the CA each of these "rows" needs many columns:

Notice the nested loops, **i** is for the rows, **j** is for the columns.

```
for(let i=0;i<yCells;i++){
    grid[i]=[];
    for(let j=0;j<xCells;j++){
      grid[i][j]=new Cell(j*5,i*5);
    }
  }
```

Each **(i,j)** location gets a new cell (of size 5)

Main Loop

Because each Cell is an object with a **show()** function, we just need to loop over all of them to draw them to the screen.

```
let currentRow=1;
function draw() {
  background(220);
  for(let i=0;i<yCells;i++){
    for(let j=0;j<xCells;j++){
      grid[i][j].show();
    }
  }
//rules below
...
}
```

Then, the rules are applied before the next frame.

Note that I keep track of the **currentRow**, this is important for updating the rules.

Main Loop

Because each Cell is an object with a **`show()`** function, we just need to loop over all of them to draw them to the screen.

```
let currentRow=1;
function draw() {
  background(220);
  for(let i=0;i<yCell
    for(let j=0;j<xCe
      grid[i][j].show
    }
  }
//rules below
...
}
```

Let's code it up, since I left out some of the details.

Then, the rules e applied before he next frame.

Note that I keep track of the **`currentRow`**, this is important for updating the rules.

Exercise: Random Ruleset

When the program states, select a random ruleset

When it reaches the bottom, restart with a random ruleset.

Topics for Today

- 1D Cellular Automata
- **2D Cellular Automata**
- Steering (if we have time)

The Game of Life

Two-Dimensions!

More Complex than 1D:
Each neighborhood is eight tiles, which leads to more configurations but also more useful applications

In 1970 Martin Gardner wrote an article in *Scientific American* that documented John Conway's "Game of Life", describing it as "recreational" mathematics. He recommended getting out a chess board and some checkers and playing for yourself.

These days, the "Game of Life" is more of a computational cliché.

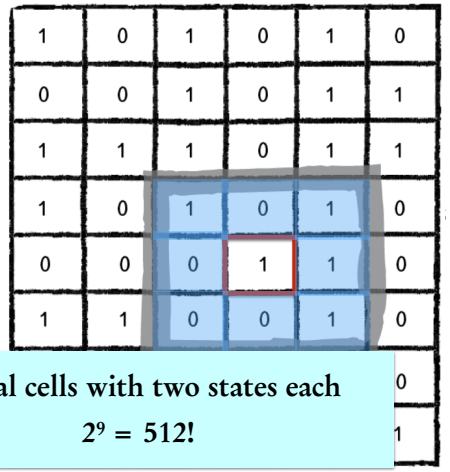
The Game of Life

Two-dimensional cellular automata

Instead of a line of cells, we have a two-dimensional matrix of cells.

Each cell only has a value of 0 or 1 (dead or alive).

The neighborhood is also large, eight total cells.



9 total cells with two states each

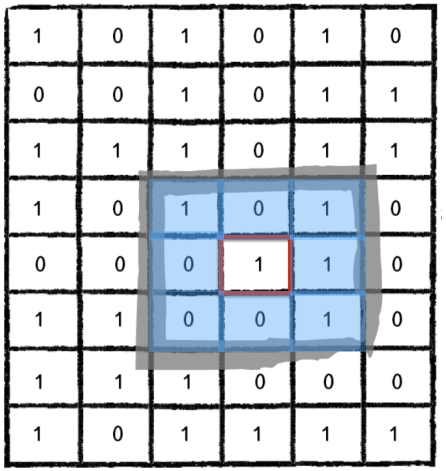
$2^9 = 512$!

The Game of Life

Two-dimensional cellular automata

Obviously we don't want to define 512 rules, so what do we do?

Let's define some rules based on the general look of the neighborhood...



The Game of Life - Rules

Death of a Cell:

Overpopulation: four or more alive neighbors

Loneliness: one or fewer alive neighbors

Birth of a Cell:

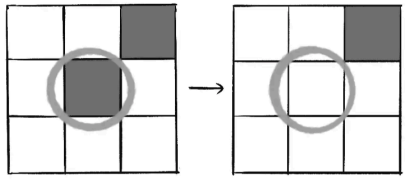
If a cell is dead it comes back to life if it has exactly 3 alive neighbors

Stasis:

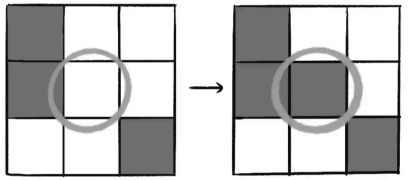
Stay Alive: Exactly two or three live neighbors

Stay Dead: Anything other than three live neighbors

The Game of Life - Rules Example



Death
(only one live neighbor)



Birth
(three live neighbors)

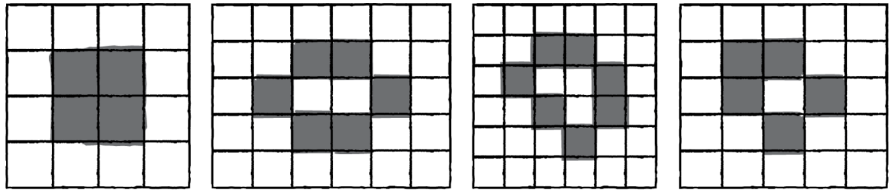
The Game of Life - Notes

For the 1D CA, we can "stack up" the generations, but that's not very viable for 2D.

The typical way to present the "Game of Life" is to treat each generation like a single frame in a movie. Instead of seeing all the generations at once, we see them over time!

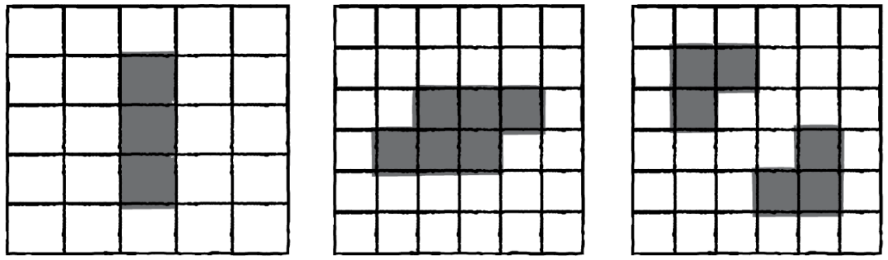
There are a few interesting structures that we will see in the simulation: some that don't change, some that oscillate back and forth between two states and some that move across the grid...

The Game of Life - Notes



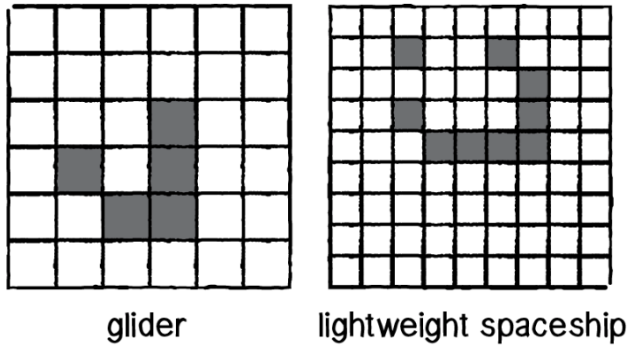
These are stationary

The Game of Life - Notes



These will oscillate between two states

The Game of Life - Notes



These will move across the grid

Implementation

- There are many similarities in implementation between 1D and 2D CA.
- Both need a 2D grid and rely on a set of rules.
- The big differences are:
  - 2D CA needs two grids, one to read from and one to write to.
  - GoL needs us to look at a larger neighborhood

We need two grids because we don't want to update "in-place" because we need to look at neighboring cells.

Implementation

- There are many similarities in implementation between 1D and 2D CA.
- Both need a 2D grid and rely on a set of rules.
- The big diff
  - 2D CA n read from
  - GoL need neighborh

We need two grids because we don't want to update "in-place" because we need to look at neighboring cells.

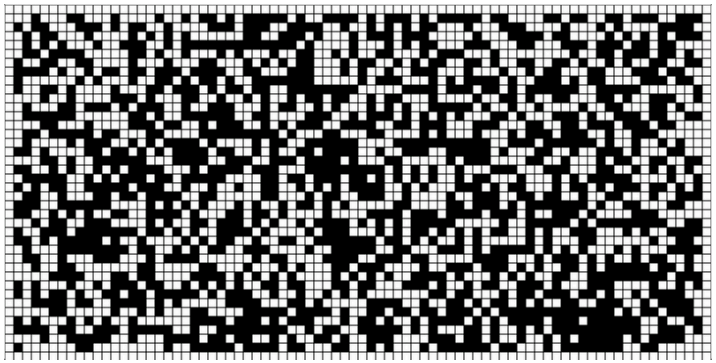
If we look at Cell `(i,j)`, these are all of the neighboring cells.

This time, we'll need four nested loops!

i - 1, j - 1	i, j - 1	i + 1, j - 1
i - 1, j	i, j	i + 1, j
i - 1, j + 1	i, j + 1	i + 1, j + 1

The Game of Life

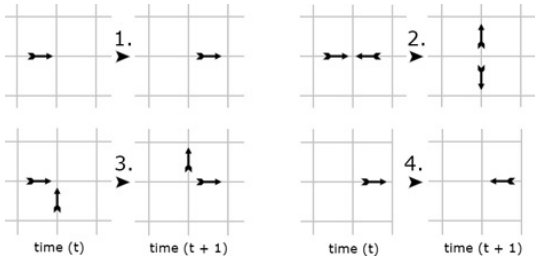
Let's code it up!



More Than Just Game of Life

CA isn't just for simple game like behavior

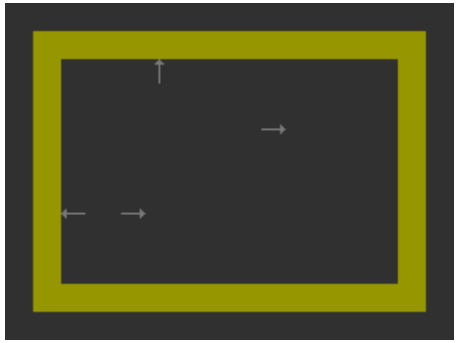
Fluid Dynamics - HPP Model:



More Than Just Game of Life

CA isn't just for simple game like behavior

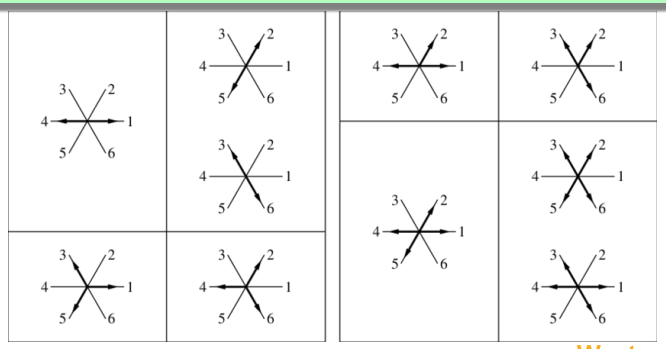
Fluid Dynamics - HPP Model:



More Than Just Game of Life

CA isn't just for simple game like behavior

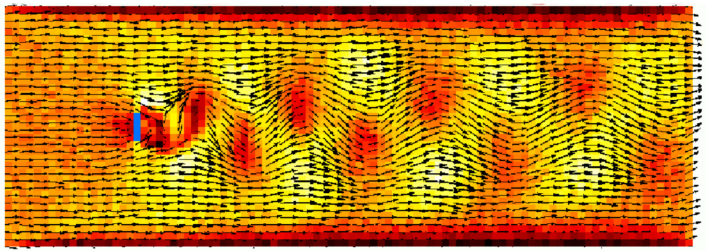
Fluid Dynamics - FHP Model:



More Than Just Game of Life

CA isn't just for simple game like behavior

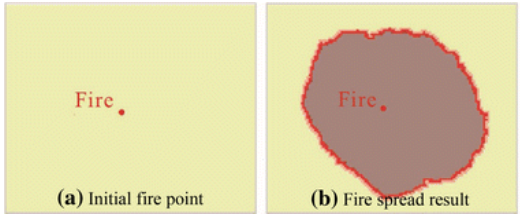
Fluid Dynamics - FHP Model:



More Than Just Game of Life

CA isn't just for simple game like behavior

Forest Fire Spread:



More Than Just Game of Life

CA isn't just for simple game like behavior

Many, Many More:

- Sand Dune Movement
- Snow Accumulation
- Disease Transmission
- Sound Wave Propagation
- Heat Diffusion
- Traffic Simulations

Topics for Today

- 1D Cellular Automata
- 2D Cellular Automata
- **Steering (if we have time)**

AI in Video Games

- Three main areas:
  - **Movement** (single entities)
  - **Decision Making** (short term, single character behavior)
  - **Strategic AI** (long timer, group behavior)
- Not all games need all areas (i.e. chess vs platforms)
- Associated Issues:
  - Gameworld interface (input to AI)
  - Animation and Physics
  - Scheduling of AI

Complexity Fallacy

https://www.youtube.com/watch?v=oO2ADGl3f4k

AI Movement

- Movement of character around the level

Input:	Output:
Geometric data about the state of the world + current position of the character and other physical properties	Geometric data representing the movement of the entity (velocity and/or acceleration)

- For most games, character have at least two states, stationary and moving

AI Movement

- Kinematic Movement: Constant velocity, no acceleration
- Steering: Dynamic movement with acceleration. Takes into account current velocity and outputs acceleration
- Examples:
  - Kinematic algorithm from A to B returns direction
  - Steering algorithm from A to B return acceleration

Kinematic Movement

- Contains four parts:
  - Position: x and y location in world space
  - Orientation: Facing
  - Linear Velocity: x and y components of velocity
  - Angular Velocity: radians per second rotation speed

```
struct Kinematic {
  Vec2 position
  float orientation
  Vec2 velocity
  float rotation
}
```

We can easily set this up as a **`struct`** or **`class`** to use as a component in entities

Kinematic Movement: Physics Lesson

- Velocity and acceleration are just first and second derivatives of position:

$$\mathbf{v}(t) = \mathbf{r}'(t) \qquad \mathbf{a}(t) = \mathbf{r}''(t)$$

- So, we can use the kinematic equations from physics to solve for velocity:

$$\mathbf{r} = \mathbf{r}_o + \mathbf{v}t + \frac{1}{2}\mathbf{a}t^2 \qquad \mathbf{v} = \mathbf{v}_o + \mathbf{a}t$$

$$\theta = \theta_o + \theta' t + \frac{1}{2}\theta'' t^2 \qquad \theta' = \theta'_o + \theta'' t$$

If the acceleration is zero, these equations are what we coded in our previous games!

Kinematic Movement: Physics Lesson

- The code for the kinematic update is very easy:

```
void update(Kinematic k, float time) {
  k.position+=k.velocity*time;
  k.orientation+=k.rotation*time;
}
```

In Assignment 3, our orientation is just left or right

With this idea in hand, let's think about the next step: steering behavior

Steering

- Uses a force to change our velocity
- Adds more realism to entity movement
- Lots of parameters to play with to get the feel/effect that you want for your entities

By adding this force, our calculations become a bit more complicated, but we get a more flexible/realistic movement

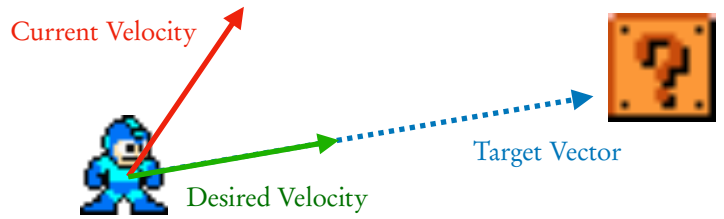
We'll see this kind of tradeoff everywhere in computing

Steering

- Having our entities change the direction instantly probably doesn't look very good except in simpler games (like Assignment 3)
- Visualize steering as applying a force to the entity that, over time, gets it to the target.
- This gives us the smooth movement that you see in many games

Steering Idea

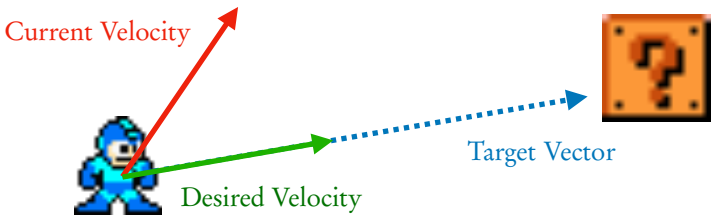
- Compute the direction to the target (same as kinematic movement)
- Change entity velocity vector to point in the desired velocity over time (not instant)



Steering Idea

- To get the **desired** velocity:

```
desired=target-pos;    //vector from player to target
desired.normalize();   //change length to 1
desired*=speed;        //change to correct length
```

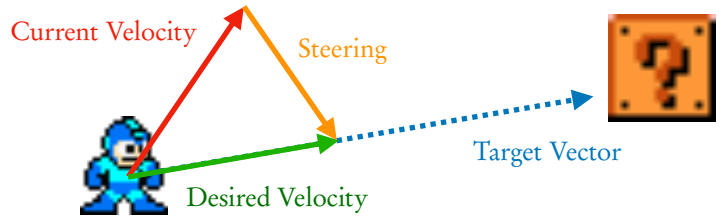


Steering Idea

- To get the **steering** vector:

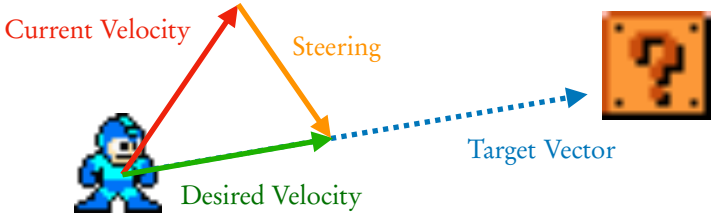
```
steering = desired-velocity
```

- The steering vector is how we get from our current velocity to the desired velocity



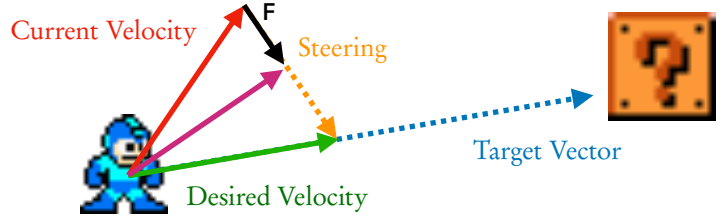
Steering Idea

- Moving from the current velocity to the desired velocity instantly is not what we want
- Instead, we only change are velocity by the steering direction using a maximum allowed acceleration



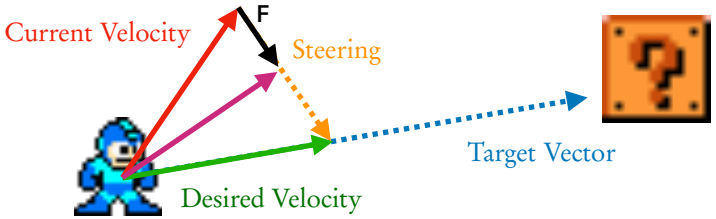
Steering Idea

- We only change our velocity by the steering direction a small amount per frame: $F = ma$
- This give us our new velocity on the next frame



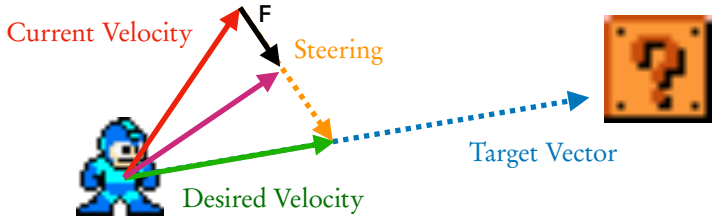
Steering Idea

- The force vector will be in the same direction as the steering vector, with some magnitude
- The magnitude is usually restricted to some max acceleration chosen by you



Steering Idea

```
maxAccel = 10.0f;                //set max allowed accel
steering = desired-velocity;     //compute steering vec
steering=steering.normalize();   //normalize steering
steering*=maxAccel;              //scale steering vec
acceleration+=steering;          //change our accel
```



Steering Idea

```
maxAccel = 10.0f;                //set max allowed accel
steering = desired-velocity;     //compute steering vec
steering=steering.normalize();   //normalize steering
steering*=maxAccel;              //scale steering vec
acceleration+=steering;          //change our accel
```

`t` is the time between frames

How do we use acceleration in our code?

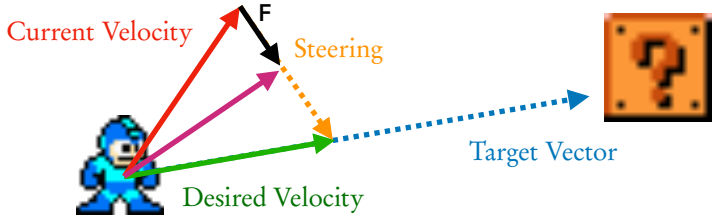
Use the full version of the kinematic equations:

$$\mathbf{r} = \mathbf{r}_o + \mathbf{v}t + \frac{1}{2}\mathbf{a}t^2 \qquad \mathbf{v} = \mathbf{v}_o + \mathbf{a}t$$

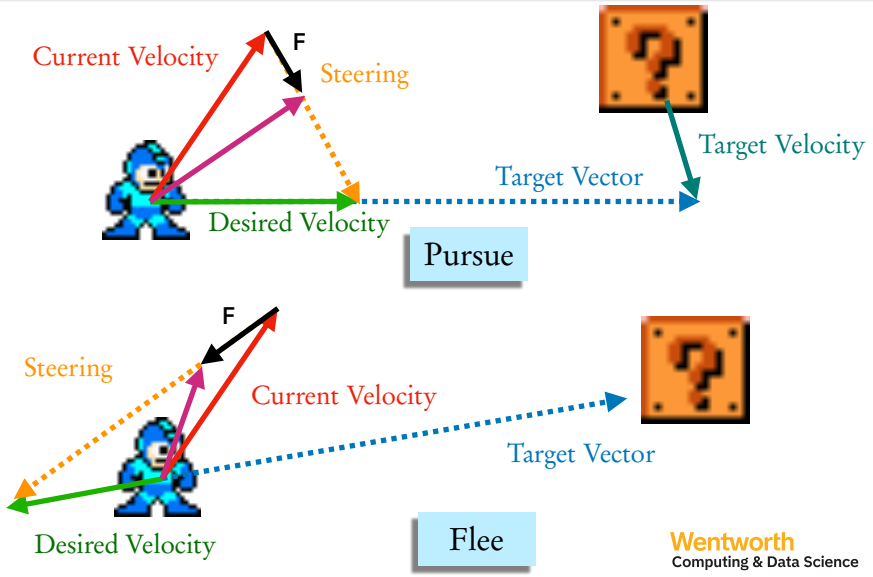
cience

Steering Idea

- This steering behavior is called “Seek”
- Works great when the target is not moving
  - In most games though, the target will probably be moving



Other Behaviors



Arriving

- If you code this up, one clear problem is what happens when we reach the target: oscillations back and forth
- So, we get the smooth movement that you see in many games, but with a problem when we get there
- Arrival behavior fixes this:
  - Choose two radii, a larger one to start slowing down and a smaller one to stop at
  - Slow down using a time and desired velocity (to compute an acceleration that slows you down appropriately)

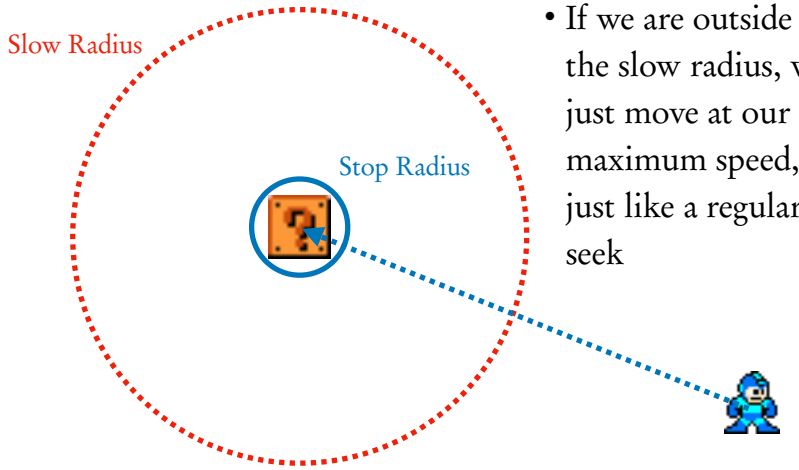
Arriving

- Very similar to seek, with a few additions:
  - If we are within the **`stopRadius`**, set velocity to zero
  - If we are within the **`slowRadius`**, scale our **`maxSpeed`** to something smaller:

```
scaledSpeed=maxSpeed*distance/slowRadius;
```

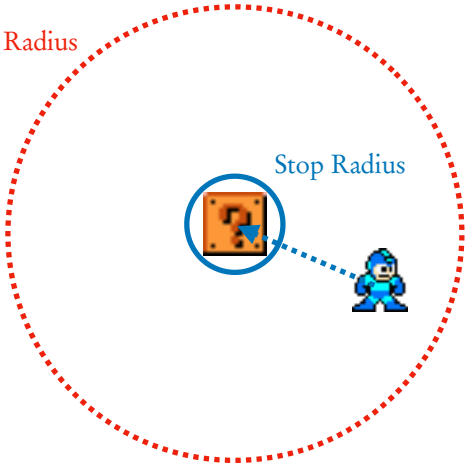
  - Compute the desired vector using the scaled speed
  - Compute the steering normally, then divide by a chosen **`timeToTargetVelocity`**

Arriving



- If we are outside of the slow radius, we just move at our maximum speed, just like a regular seek

Arriving

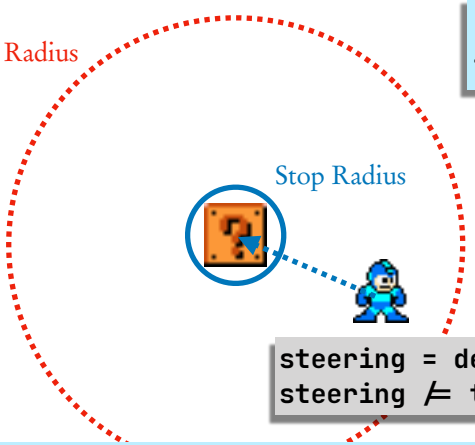


- If we are inside the slow radius, scale our desired speed by how far in we have traveled

This sets our desired vector magnitude (speed) to something smaller than we are now, slowing us down

```
scaledSpeed=maxSpeed*distance/slowRadius;
```


Arriving



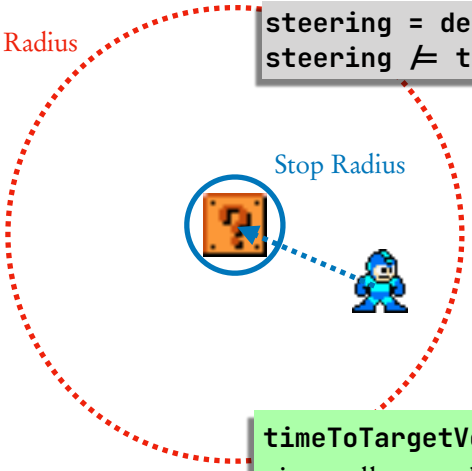
What about `timeToTargetVelocty`?

This represents the time is should take to change our velocity to the target velocity

```
steering = desired-velocity;
steering /= timeToTargetVelocity;
```

Dividing the steering (our update to acceleration) by this value gives us a larger acceleration when `TTTV` is less than 1

Arriving




```
steering = desired-velocity;
steering /= timeToTargetVelocity;
```

As long as the acceleration is not larger than our max acceleration, we get to the target velocity in a short time

`timeToTargetVelocity` is usually a small value, like 0.1

Extra Resources

- RedBlobGames:
  - Path finding, A*, graph theory
  - https://www.redblobgames.com/
- Nature of Code
  - Many topics including autonomous agents and steering (slightly different than what we did today)
  - https://natureofcode.com/