Simulation – Model, View, Controller

15-110 – Wednesday 04/05

New Ice Creams I Learned About

- **Badam Kulfi** – an almond Indian cold desert
- **Bavarian Chocolate** – yum
- **Berry –** mixed berries I assume? nice!
- **Brownie Brittle –** I like soft brownies personally, but you do you
- **Cajeta –** like caramel or dulce de leche, but silkier
- **Cheese –** it's a real thing from the Philippines!
- **Cinnamon** – I've seen cinnamon apple, cinnamon bun, and cinnamon cheesecake, but never just cinnamon!
- **Coffee Cookies and Cream** – both parts are common, but not together!
- **Cookies and S'mores –** see right above
- **Durian –** now I want to try this!
- **Eggnog** – delicious
- **Lil' Blue Panda** – sugar cookie flavored blue and white ice cream
- **Mocha Almond Fudge** – nice combo
- **Oat of this Swirled –** like cinnamon oatmeal cookies
- **Peanut Butter Chocolate Chip –** delicious

Announcements

- **Tutorial:** how to work with Hw6 starter files

Learning Goals

- Represent the state of a system in a **model** by identifying **components** and **rules**
- **Visualize** a model using graphics
- Update a model over **time** based on **rules**

Simulations and Models

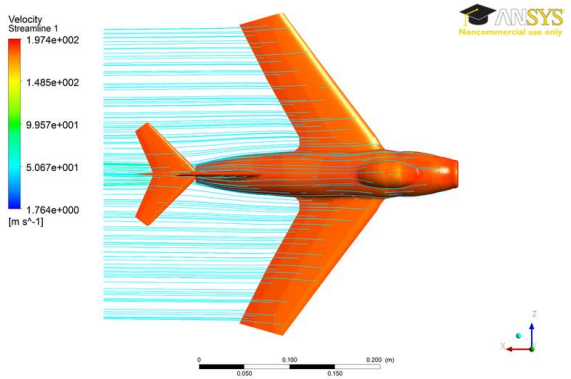
Simulations are Imitations of Real Life

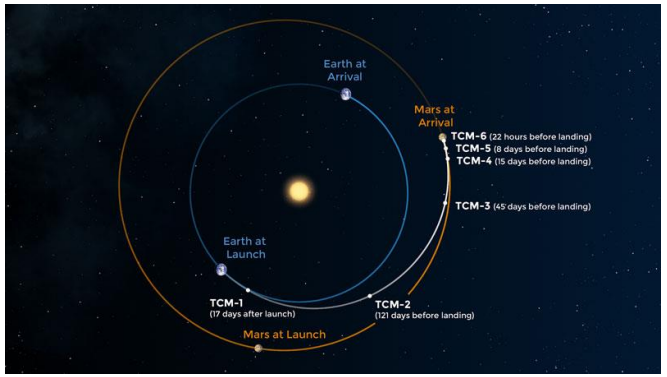
A **simulation** is an automated imitation of a real-world event.

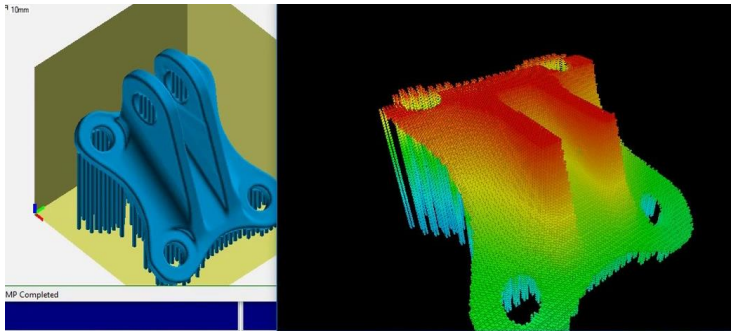
By running simulations on different starting inputs, and by interacting with them while they run, we can test how the event will change under different circumstances and learn interesting things.

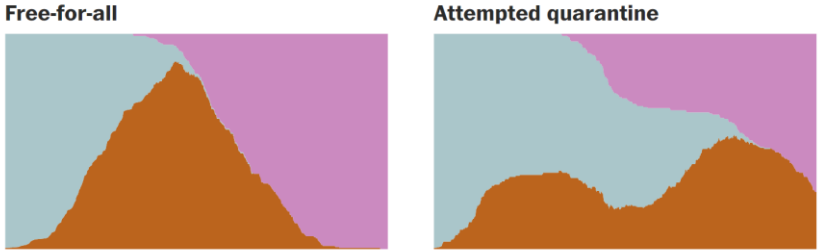
Examples of Simulations

Simulation is used across many different fields, including training people, testing designs, and making predictions (like whether a flight plan will work, or how a pandemic will evolve over time).









Simulations vs. Real-world Experiments

Simulations share a lot in common with real world experiments. Major differences include:

- Experiments run in **real time**; simulations can be **sped up, slowed down, or paused**.
- Experiments can be **expensive**; simulations are fairly **cheap**.
- Experiments include **all possible factors**; simulations only include **factors we program in**.

Example Simulations

You can explore simulations across a variety of fields on the site NetLogo, which is focused entirely on modeling and simulation.

- Ant colony movements
- Flocking behavior
- Gravitational forces
- Climate change
- Fire spreading
- Rumor mills

Simulations Run on Models

How do we program a simulation? You need to design a good **model**, which will mimic the part of the real world you want to study. The simulation showcases how the system represented by the model changes **over time**, or how it changes **based on events**.

Models are composed of two parts:

- The **components** of the system (information that describes the world at an exact moment).
- The **rules** of the system (how the components should change as time passes/events occur).

Components are like variables, and rules are like functions!

Example Model

Problem: how much ice cream can an ice cream shop expect to sell on a given day?

Model Components: price; temperature; number of flavors available

Model Rules: some number of people buy ice cream every day; when it is hotter outside, people buy more ice cream; when prices go up, demand goes down; an increased number of flavors increases the number of people who buy ice cream (to a certain point)

Activity: Design a Model

Problem: we want to track how many birds are in a local area over the course of a year, to see how the population changes.

You do: What are the components of this model? What are the rules?

Important: Simulations Rely on the Model!

Simulations are powerful, but they can also be suspect to error and bias, because **the results are influenced by what is included in the model**.

Example: you could build a fancy simulation of an amusement park to test different park configurations and estimate how much profit could be expected from each arrangement. But if your model doesn't include a variable for weather, the results may all be overly-optimistic.

Try investigating any simulations you might interact with to see what biases and errors they might include.

Coding a Simulation

Simulation Parts in Code

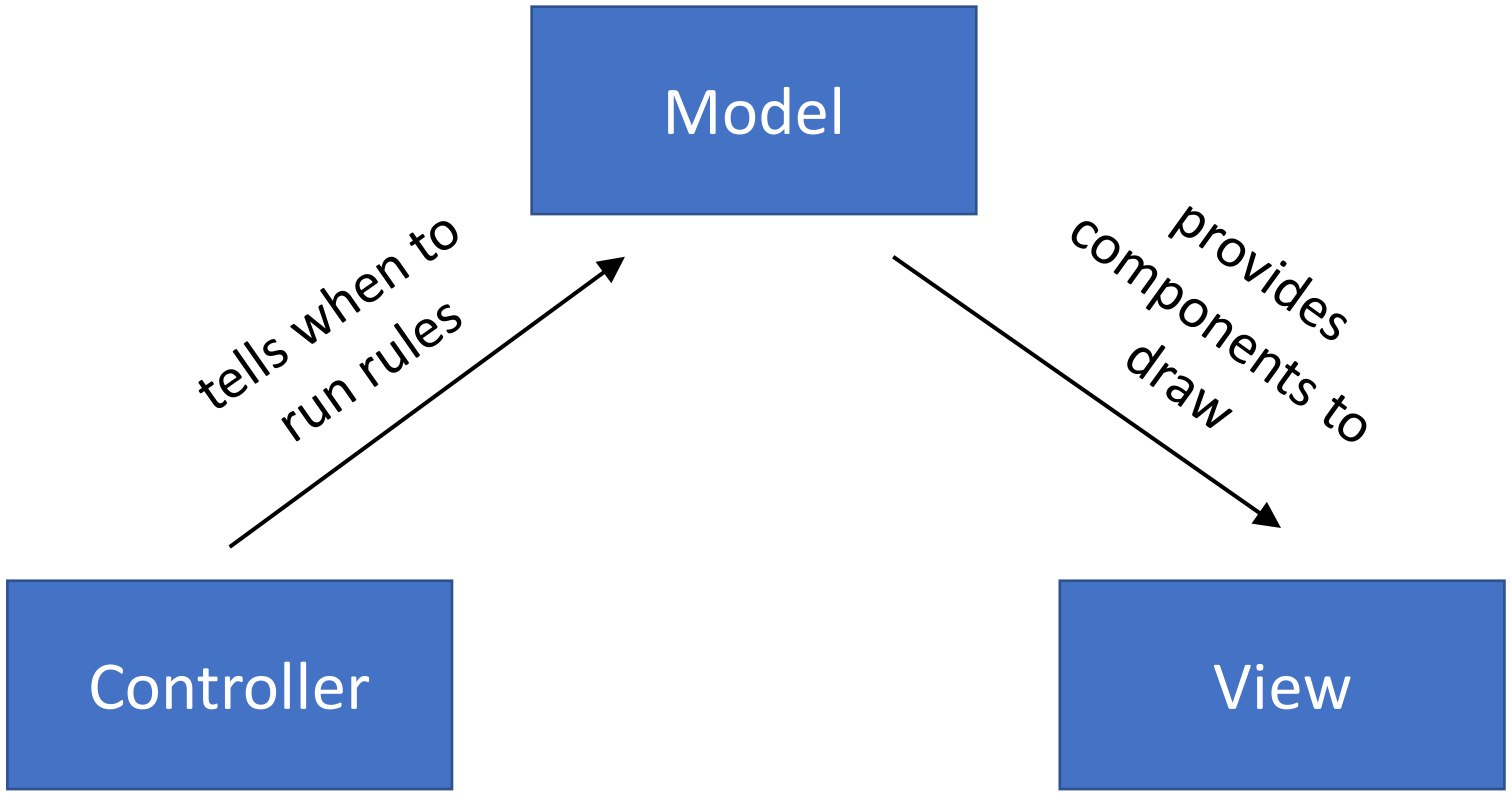
We'll implement simulations in this class **graphically**, like in NetLogo, using Tkinter.

We'll start with very abstract simulations (to keep the code simple) and will show how to program more complex simulations next time.

Our simulation code will be composed of three parts:

- A **model** which stores the core components in a shared data structure and implements core rules in functions
- Time and event **controllers** which tell the model when to run rules that update the components
- A graphical **view** which repeatedly displays the current state of the model

Model, View, Controller



Making the Components

We'll represent the model's components in code in a **dictionary** called `data`. The keys will take the place of variable names and the values will be the actual component values.

For example, to store information about a circle that represents some part of the model, we could set:

```
data["x"] = 200
data["y"] = 200
data["r"] = 50
```

Storing all the components in one structure lets us pass the same structure around to all the functions we write using **aliasing**. This will let us update components in a rule function, then display the same (updated) data in a separate view function.

Displaying the Model

To display the whole model, we'll use Tkinter to draw graphics that represent the components visually. By referring to component values in `data` in the view function, we can make graphics that change alongside the model.

For example, if `data = { "x" : 200, "y" : 200, "r" : 50 }`, we could draw a circle with:

```
canvas.create_oval(data["x"] - data["r"], data["y"] - data["r"],
                   data["x"] + data["r"], data["y"] + data["r"])
```

We'll erase and re-draw the graphics window every time the rules of the simulation run. If we change the components a little bit at a time, this makes the display appear to update smoothly.

Running the Rules

We can run the simulation rules in two ways: either **over a period of time** or **when events happen** (or both!). We'll address the time controller first, then the event controller in a later lecture.

The **time controller** will create a **time loop** and call a function that implements the model's rules within that time loop at equal time intervals. By calling this function continuously, we can simulate time passing.

If the model's rules change the model's components in `data`, this will simulate the model changing over time!

```
data["x"] = data["x"] + 5 # move the circle to the right
```

Simulation Functions

We'll use a new **simulation framework** that you can find linked on the course website to support our simulations. This framework manages the controllers for you; you just need to focus on implementing the model and the view. To do this, update three functions to build a simple simulation:

- `makeModel(data)` makes the original components. `data` is the model dictionary
- `runRules(data, call)` runs the rules to update `data`. The integer `call` represents the number of times `runRules` has been called
- `makeView(data, canvas)` displays the model. `canvas` is a Tkinter canvas

This is different from the code we're used to because the functions **work together** instead of running in a sequential order.

Sidebar: Controller Functions – Time Loop

The starter code we provide helps the simulation run smoothly. You don't need to understand this code, but here's more details if you're interested.

The **time** controller in the function `timeLoop` calls our function `runRules`, then calls `makeView` to update the view. It simulates a time loop with the built-in function `canvas.after`. This function calls `timeLoop` again (like recursion) but pauses before making the call. That lets us recurse infinitely without freezing the window.

The function `runSimulation(width, height, timeRate)` sets up this time loop. You can speed up/slow down the simulation by changing `timeRate` in the function call.

You can also change the window size by changing `width` and `height` in the function call arguments.

Simple Example – Color-Changing Ball

Let's start with a simple simulation. Say we want to draw a circle and have the color of the circle change over time.

The **components** should hold any values that might change. In this case, that's the **color** of the circle. Set an initial component value in `makeModel`.

The **rules** should describe how the model changes over time. In this case, we **change the color** in the shared dictionary with every call to `runRules`.

The **view** should draw a circle in the middle of the window and set its color based on the color in the model. This is done in `makeView`.

Simple Example Code

```
def makeModel(data):
    # put variables in data here
    data["color"] = "red"

def makeView(data, canvas):
    # (200, 200) is center point
    # make sure to reference data for the parts that change!
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,
                       fill=data["color"])

def runRules(data, call):
    import random
    # Let's pick a color randomly!
    newColor = random.choice(["red", "orange", "yellow",
                              "green", "blue", "purple"])
    data["color"] = newColor # update data to change the model
```

Activity: Make the circle grow

You do: open the simulation starter code and copy in the functions from the previous slide. Run the code to make sure it works, then modify the code in the three functions so that the circle also **grows larger** as time passes.

Hint: you'll need to add one **component** to the model, the thing that is changing. You should change that component in `runRules` and access it while drawing the circle in `makeView`.

Summary: Model, View, Controller

Throughout the process of building simulations, we've structured code based on the **model, view, controller** framework.

Model: manages the components and rules of the thing we're simulating

View: displays the data in the model so that the user can look at it

Controller: manages time loops and events that provide changes to the model

Learning Goals

- Represent the state of a system in a **model** by identifying **components** and **rules**
- **Visualize** a model using graphics
- Update a model over **time** based on **rules**